异步图书
www.epubit.com.cn

MANNING

概率编程实战

Practical Probabilistic Programming

〔美〕Avi Pfeffer 著
姚军 译

加州大学伯克利分校计算机科学教授
Stuart Russell 作序


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

MANNING

概率编程实战

Practical Probabilistic Programming

〔美〕Avi Pfeffer　著
姚军　译

人 民 邮 电 出 版 社
北 京

异步社区
人 民 邮 电 出 版 社
www.epubit.com.cn

异步社区电子书

感谢您购买异步社区电子书！异步社区已上架电子书 500 余种，社区还会经常发布福利信息，对社区有贡献的读者赠送免费样书券、优惠码、积分等等，希望您在阅读过程中，把您的阅读体验传递给我们，让我们了解读者心声，有问题我们会及时修正。

社区网址： http://www.epubit.com.cn/

反馈邮箱： contact@epubit.com.cn

异步社区里有什么？

图书、电子书（半价电子书）、优秀作译者、访谈、技术会议播报、赠书活动、下载资源。

异步社区特色：

纸书、电子书同步上架、纸电捆绑超值优惠购买。

最新精品技术图书全网首发预售。

晒单有意外惊喜！

异步社区里可以做什么？

博客式写作发表文章，提交勘误赚取积分，积分兑换样书，写书评赢样书券等。

联系我们：

微博：

@ 人邮异步社区　　　@ 人民邮电出版社 - 信息技术分社

微信公众号：

人邮 IT 书坊　　　异步社区

QQ 群： 368449889

图书在版编目（ＣＩＰ）数据

概率编程实战 / (美) 艾维·费弗 (Avi Pfeffer)
著 ; 姚军译. -- 北京 : 人民邮电出版社, 2017.4
ISBN 978-7-115-44874-3

Ⅰ. ①概… Ⅱ. ①艾… ②姚… Ⅲ. ①概率统计一程
序设计 Ⅳ. ①O211-39

中国版本图书馆CIP数据核字(2017)第036152号

版权声明

♦ 著　　[美] Avi Pfeffer
译　　姚　军
责任编辑　王峰松
责任印制　焦志炜

♦ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市海波印务有限公司印刷

♦ 开本：800×1000　1/16
印张：24.25
字数：520 千字　　2017 年 4 月第 1 版
印数：1 – 2 500 册　　2017 年 4 月河北第 1 次印刷

著作权合同登记号　图字：01-2016-3947 号

定价：89.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

内容提要

概率推理是不确定性条件下做出决策的重要方法，在许多领域都已经得到了广泛的应用。概率编程充分结合了概率推理模型和现代计算机编程语言，使这一方法的实施更加简便，现已在许多领域（包括炙手可热的机器学习）中崭露头角，各种概率编程系统也如雨后春笋般出现。本书的作者 Avi Pfeffer 正是主流概率编程系统 Figaro 的首席开发者，他以详尽的实例、清晰易懂的解说引领读者进入这一过去令人望而生畏的领域。通读本书，可以发现概率编程并非“疯狂科学家”们的专利，无需艰深的数学知识，就可以构思出解决许多实际问题的概率模型，进而利用现代概率编程系统的强大功能解题。本书既可以作为概率编程的入门读物，也可以帮助已经有一定基础的读者熟悉 Figaro 这一概率编程利器。

序

1814 年，皮埃尔•西蒙•拉普拉斯写道，“在很大程度上，人生最重要的问题就是概率问题。”此后过了 100 多年，回答这些问题（这一格言依然正确）的唯一方法是用笔和纸分析每个问题，得到结果的公式，手工填入数字以求得公式值。计算机的出现对这一情况并没有很大的改变，只是能够为包含更多数字的更复杂公式求值，纸笔分析也变得更加雄心勃勃，往往用纸数百页。

概率问题的分析需要构思**概率模型**，这种模型以某种方式规划概率空间，为其指定数值化概率。过去，概率模型用自然语言文本和半正式的数学标记法的组合写下。从模型中，经过进一步数学处理得出计算答案的公式或者算法。这些阶段都十分费时费力、容易出错，而且存在特定于具体问题的难点，使概率理论的适用性受到了严重的限制。尽管拉普拉斯在多年前就已提出，但是这个生活中最重要的问题仍然没有答案。

解决上述问题的第一个重要进步是定义概率模型所用的**形式语言**的发展，例如贝叶斯网络和马尔科夫网络。形式语言具有定义正确表达式的精确语法，以及定义每种正确表达式含义的精确语义（即每个表达式表示哪种概率模型）。因此，用机器可理解的形式描述概率模型，开发一个算法计算任何可表达概率模型结果都成为可能。

在前面的叙述中，美中不足的是：**可表达**概率模型的缺乏。实际上，贝叶斯和马尔科夫网络等形式语言表达能力相当有限。从某种意义上说，它们只是布尔电路的概率模拟。为了对这一局限性的含义有所了解，我们考虑一个问题：编写大型公司所用的工资单软件。在 Java 等高级编程语言中，这可能涉及数万行代码。现在，想象一下将许多逻辑门电路连接起来完成相同的功能。这样的任务似乎完全无法完成。这样的电路规模、复杂度和清晰性都无法想象，因为电路缺乏捕捉问题结构的表达能力。

1997 年，本书作者 Avi Pfeffer（当时还是个学生）和他的导师 Daphne Koller 以及协作者 David McAllester 发表了一篇关于概率编程语言（PPL）的原创论文，提供了将概率理论与高级编程语言联系起来的关键思路。这一思路是通过引入随机元素使程序成

为概率模型，并将程序的意义定义为每个可能执行路径的概率。这一思路以高效的方式结合了数学的两个最重要分支，我们接下来将要开始探索由此产生的新可能性。

本书使用 Figaro 语言阐述这些概念及其应用，逐步引领读者理解上述思路。书中避开了不必要的数学知识，集中于详细构思、认真解释的实例，适合于拥有典型编程背景的读者。通读本书还有一个副产品：读者能够比以往更轻松地熟练掌握贝叶斯推理和统计学的原理及技术。最重要的是，读者将学习建模技能，这是任何科学家或者工程师的最关键技能之一。Figaro 和其他 PPL 使人们可以直接、快速、精确地表现这种技能。

本书是将概率编程从开发它的实验室中转移到真实世界的重要步骤。从某种程度上说，PPL 系统的能力无疑还难以应对这种挑战，那些研究实验室也将停止工作。另一方面，本书的读者一定能找出应用 Figaro 的创新方法，它与各种新问题的相关性也绝非作者所能想象的。

Stuart Russell
加州大学伯克利分校计算机科学教授

前言

概率编程是一个激动人心的新领域，正在快速地引起人们的兴趣，从学术领域进入程序员的世界中。本质上，概率编程是创建概率推理模型的新方法，这种模型用来根据观测预测或者推理未知的事物。概率推理很久以来都是机器学习的核心方法之一，在机器学习中，使用了概率模型来描述从经验中得到的知识。在概率编程之前，概率推理系统局限于包含贝叶斯网络等简单、固定结构的模型。而概率编程提供了编程语言的全部能力以表现模型，使概率推理系统摆脱了这些桎梏。这正如从电路转向高级编程语言。

我从青少年时代用 BASIC 语言开发一个足球模拟程序时就致力于概率编程，只是当时还没认识到。这个模拟程序使用“GOTO 1730 + RANDOM * 5”这样的指令表示随机的事件顺序。经过精心的调校，模拟程序已经很逼真，足以让我娱乐数个小时。当然，在随后的岁月中，概率编程已经逐渐成熟，不再只是包含随机目标的 GOTO 语句了。

1997 年，我和 Daphne Koller、David McAllester 合作编撰了第一篇关于概率编程的论文。这篇论文引入了一种类似 Lisp 的概率语言，但是主要的创新是根据关于输出的证据，推理程序可能特征的一种算法。这一创新不仅提供了运行程序以获得可能执行方式的手段，还反向论证和推理了产生观测结果的原因，从而使概率语言超越了典型的概率模拟语言。

21 世纪初，我开发了第一种基于函数式编程的通用概率编程系统 IBAL。IBAL 有很强的表达能力并包含新型推理算法，但是几年之后，我逐渐对其局限性感到不满，主要是难以与数据交互、与应用程序集成。这些局限性促使我在 2009 年开始开发新的概率编程系统，我将其定名为 Figaro。Figaro 以实用性作为首要目标，同时并没有牺牲概率编程能力。这导致了将 Figaro 作为 Scala 程序库的设计决策，该决策使得概率编程模型更容易与 Java 虚拟机应用集成。同时，Figaro 具备了我所知的概率编程系统中最广泛的表现特征和推理算法。Figaro 现在是一个开源 GitHub 项目，最新版本号为 3.3。

概率编程可能是一种难以掌握的技术，因为它需要多种技能，其中主要的是编写概

率模型和编写程序的能力。对于许多程序员来说，编写程序很自然，但是概率建模有些神秘。本书的目的是揭开概率建模的神秘面纱，告诉您如何在创建概率模型时高效编程，帮助您有效地使用概率编程系统。本书假定读者在机器学习或者概率推理上没有任何背景。函数式编程和 Scala 的经验有所帮助，但是要使用本书并不一定要成为 Scala 的奇才，Scala 专业知识也可能因为阅读本书而增长。

阅读本书之后，您应该可以在没有机器学习博士学位的情况下，为许多应用程序设计概率模型，从数据中获得有意义的信息。如果您是某个领域的专家，本书能够帮助您表达脑海中或者纸面上的模型，使它们可以运算，实现对不同概率的计算和分析。如果您是一位数据科学家，本书可以帮助您开发比其他工具更丰富、更详细和更精确的模型。如果您是软件工程师或者架构师，正寻求在系统中加入不确定情形下的推理能力，本书不仅能够帮助您构建处理不确定性的模型，还能将这些模型集成到应用程序中。不管因为何种原因选择本书，我都希望您能够喜欢它，并从中得益。

致谢

本书的创作经过了许多年：从关于概率编程的第一个思路，到 IBAL 和 Figaro 系统的创建，再到构思、编写并与 Manning 出版社一起完善。在这段时间里，许多人贡献了自己的力量，使本书得以面世。

本书的出版很大程度上归功于我在 Charles River Analytics 的团队的努力：Joe Gorman、Scott Harrison、Michael Howard、Lee Kellogg、Alison O'Connor、Mike Reposa、Brian Ruttenberg 和 Glenn Takata。还要感谢 Scott Neal Reilly 从一开始就支持 Figaro。

在人工智能和机器学习方面，给我最大教益的是我的导师和合作者 Daphne Koller。Stuart Russell 为我提供了学习人工智能的第一个机会，在整个职业生涯中鼓励我，并成为最新的合作者和本书的序言撰写者。Mike Stonebraker 在其 Postgres 项目中为我提供了第一个研究机会，在他的小组中工作时，我学到了许多关于系统构建的知识。Alon Halevy 曾邀请我和他在 AT&T 实验室中一起度过一个夏季，我在那里第一次和 David McAllester 讨论关于概率编程的问题，成果就是和 Daphne 合作编写的 Lisp 概率论文。当这些思路刚刚萌芽时，我总是和我的合作者和同事 Lise Getoor 一起探讨。

我深深感谢 Alex Ihler，他慷慨地贡献自己的专业知识，认真阅读本书以审核技术上的准确性。过去几年中，在所有与推理相关的事情上，Alex 总是极好的意见反馈者。

在不同的发展阶段，还有其他许多人提供了意见，包括 Ravishankar Rajagopalan、Shabeesh Balan、Chris Heneghan、Clemens Baader、Cristofer Weber、Earl Bingham、Giuseppe de Marco、Jaume Valls、Javier Guerra Giraldez、Kostas Passadis、Luca Campobasso、Lucas Gallindo、Mark Elston、Mark Miller、Nitin Gode、Odisseyas Pentakolos、Peter Rabinovitch、Phillip Bradford、Stephen Wakely、Taposh Dutta Roy 和 Unnikrishnan Kumar。

感谢 Manning Publications 的许多出色员工对本书出版的帮助。特别要感谢编辑 Dan Maharry 使这本书远远超过了我自己完成的质量，还要感谢 Frank Pohlmann 鼓励我编写本书，并且帮助我准备写作过程。

感谢空军研究实验室（AFRL）和国防部高级研究计划署（DARPA）对本书所描述的先进机器学习概率编程（PPAML）项目中某些工作的投资。特别要感谢几位 DARPA 项目经理，Bob Kohout、Tony Falcone、Kathleen Fisher 和 Suresh Jagannathan，他们对概率编程深信不疑并致力于实现它。

最后，如果没有家人的爱和支持，本书也不可能出版。感谢我的妻子 Debby Gelber 和孩子们（Dina、Nomi 和 Ruti），你们都是了不起的人。永远感谢我的母亲 Claire Pfeffer 用自己的爱养育了我。谨以本书献给你们。

关于本书

不管商业、科学、军事上还是日常生活中，许多决策都涉及不确定情况下的判断。当不同的因素将您引向不同方向，如何知道最应该注意的是哪个方面？概率模型可以表达关于您所处情况的所有相关信息。概率推理使用这些模型确定对决策影响最大的变量的概率。您可以使用概率推理**预测**最可能发生的情况：您的产品能否在目标价格上取得成功；患者对特定疗法的反应是否良好；您的候选人如果采用某种立场，能否赢得选举？您还可以使用概率推理**推导**出所发生情况的可能原因：如果产品失败，是不是因为价格太高？

概率推理也是机器学习的主要方法之一。您在概率模型中编码关于所在领域的初始信念，如用户对市场产品的一般反应。然后，提供训练数据（可能与特定产品的用户反应有关），更新信念以获得新模型。现在，可以使用新模型预测未来的结果，如规划中的产品是否成功，或者推导出观测结果的可能原因，如新产品失败的原因。

过去，概率推理使用专用语言表示概率模型。近年来，我们意识到可以使用常规的编程语言，这造就了概率编程。概率编程有三大好处。首先，在构建模型时，可以从编程语言的所有特征中获益，如丰富的数据结构和控制流。其次，概率模型很容易与其他应用程序集成。第三，可以从用于论证模型的通用推理算法中获益。

本书的目标是提供在日常活动中使用概率编程的知识。特别是：

- 如何构建概率模型并以概率程序表达。
- 概率推理的工作原理以及如何以各种推理算法实现。
- 如何使用 Figaro 概率编程系统构建实用的概率程序。

Figaro 以 Scala 程序库的形式实现。和 Scala 一样，Figaro 结合了函数式和面向对象编程风格。这对于不了解函数编程的人来说很有用。本书不使用高级函数式编程概念，所以您应该能在对此了解有限的情况下理解。同样，对 Scala 有所了解是有益的。尽管本书中常常会解释 Scala 的结构，但不是 Scala 的简介。同样，本书通常不使用 Scala 较为难懂的功能，所以略有涉猎就应该足够了。

路线图

本书的第 1 部分简介概率编程和 Figaro。第 1 章首先解释概率编程的定义及其实用性，然后简单介绍 Figaro。第 2 章是 Figaro 的使用教程，帮助您很快地了解概率程序的编写。第 3 章提供了一个完整的概率编程应用——一个垃圾邮件过滤器，包括论证给定电子邮件是常规邮件还是垃圾邮件的组件，以及从训练数据学习概率模型的组件。第 3 章的目标是在详细介绍建模技术之前，提供各种技术相互融合的全貌。

第 2 部分介绍概率程序的构建。第 4 章包含有关概率模型和概率程序的基本材料，这对理解它们，真正了解创建概率程序时需要做什么很重要。第 5 章提供了两种作为概率编程核心的建模框架——贝叶斯网络和马尔科夫网络。第 6～8 章描述了一组用于构建更高级程序的实用编程技术。第 6 章讨论使用 Scala 和 Figaro 集合组织涉及许多同类变量的程序的方法。第 7 章讨论面向对象编程，这种方法对于概率编程和常规程序同样有益。第 8 章介绍建模动态系统。动态系统是状态随时间推移而变化的系统，是这一章深入介绍的概率推理极其常见和重要的应用。

第 3 部分向您传授关于概率推理算法的知识。理解推理对于有效使用概率编程很重要，这样您就可以使用适合于任务的算法，对合适的方式配置，以支持有效推理的方式表达模型。第 3 部分在传授算法理论和使用这些算法的实践技巧之间达成了平衡。第 9 章是基础，介绍了捕捉概率推理中使用的主要思路的 3 条原则。第 10 章和第 11 章描述了两个主要的推理算法家族。第 10 章描述因子分解算法，包括对因子及其工作原理的介绍，以及变量消除和置信传播算法。第 11 章介绍抽样算法，特别关注重要性抽样和马尔科夫链蒙特卡洛算法。第 10 章和第 11 章专注于计算感兴趣的变量概率的基本查询，而第 12 章介绍如何使用因子分解和抽样算法计算其他查询，如多变量联合概率、变量最大可能值和观测证据的概率。最后，第 13 章讨论两个高级而重要的推理任务：监视随时变化的动态系统，从数据中学习概率模型的数值参数。

每章都有一组练习，涵盖了从简单计算、编程任务到开放思维练习的范围。

本书还包括两个附录。附录 A 是 Figaro 的安装指南。附录 B 是其他概率编程系统的概况。

关于代码和练习

本书的代码以等宽字体显示，以便和正文分开。许多代码清单中含有代码注释，强调了重要的概念。在某些情况下，清单之后有链接到解释的编号项目。

本书包含许多代码示例，其中大部分可以从本书网站 www.manning.com/books/practical-probabilistic-programming 的在线代码库中找到。该网站还包含部分练习答案。

关于作者

Avi Pfeffer 是概率编程的先驱，从一开始就活跃于这个领域。Avi 是 Figaro 的首席设计者和开发者。在 Charles River Analytics，Avi 参与了 Figaro 在多个问题上的应用，包括恶意软件分析、汽车健康监控、气象模型建立和工程系统评估。

在闲暇时，Avi 是一位歌手、作曲家和音乐制作人。他和妻子及三个孩子在马萨诸塞州坎布里奇生活。

作者在线

购买本书就可以免费访问 Manning Publications 运营的一个私有网络论坛，在那里可以评论本书，提出技术问题，讨论书中的练习，从作者和社区那里得到帮助。在 www.manning.com/books/practical-probabilistic-programming 可以访问和订阅该论坛。这个页面提供了关于注册后如何访问论坛、论坛提供的帮助类型以及行为准则的信息。

Manning 对读者的承诺是，提供读者之间和读者与作者之间有意义对话的途径。我们不能承诺作者的参与度，他们对论坛的贡献完全是自愿（无偿）的。我们建议读者向作者提出挑战性的问题，以免他们失去兴趣！

只要本书仍在销售中，作者在线论坛和过去讨论的存档都可以在 Manning 网站上访问。

关于封面

本书封面上的插图题为“威尼斯人”。这幅插图取自一本法国旅游图书——J. G. St. Saveur 于 1796 年出版的《旅游百科全书》。当时，旅游消遣还是相当新颖的现象，这样的旅游指南很受欢迎，它向旅游者和空谈旅游家介绍了法国和海外其他地区的风土人情。

《旅游百科全书》中丰富的插图生动地讲述了 200 年前世界各个城市和地区的独特个性。当时，在两个距离仅为几十英里的地区，人们的穿着就足以独特地反映所属地区。这本旅游指南展示了当时与其他历史时代（除了快节奏的现在）的孤立感和距离感。

当时的着装规范已经变化，各个地区的多样化也逐渐消失。现在，往往难以分辨不同大陆的居民。从乐观的角度看，我们用文化和视觉上的多样性换来了更多彩的个人生活——或者更丰富、有趣的知识和技术生活。

Manning 通过复活这本旅游指南中的插图，用两个世纪前丰富多彩的地域性差别赞美计算机行业的创造性和乐趣。

目录

第 1 部分　概率编程和 Figaro 简介

1

第 1 章　概率编程简介　3

1.1　什么是概率编程　4
1.1.1　我们如何做出主观判断　4
1.1.2　概率推理系统帮助决策　5
1.1.3　概率推理系统有 3 种方式推理　7
1.1.4　概率编程系统：用编程语言表达的概率推理系统　11
1.2　为什么使用概率编程　14
1.2.1　更好的概率推理　14
1.2.2　更好的模拟语言　15
1.3　Figaro 简介：一种概率编程语言　16
1.4　小结　23
1.5　练习　24

2

第 2 章　Figaro 快速教程　25

2.1　Figaro 简介　25
2.2　创建模型和运行推理：重回 Hello World　27
2.2.1　构建第一个模型　28
2.2.2　运行推理和回答查询　29
2.2.3　构建模型和生成观测值　29
2.2.4　理解模型的构建方法　31
2.2.5　理解重复的元素：何时相同，何时不同　32
2.3　使用基本构件：原子元素　33
2.3.1　离散原子元素　34
2.3.2　连续原子元素　35
2.4　使用复合元素组合原子元素　37
2.4.1　If　38
2.4.2　Dist　39
2.4.3　原子元素的复合版本　39
2.5　用 Apply 和 Chain 构建更复杂的模型　40
2.5.1　Apply　41
2.5.2　Chain　43
2.6　使用条件和约束指定证据　46
2.6.1　观测值　46
2.6.2　条件　47
2.6.3　约束　48
2.7　小结　50
2.8　练习　51

3

第 3 章　创建一个概率编程应用程序　53

3.1　把握全局　53
3.2　运行代码　56
3.3　探索垃圾邮件过滤应用的架构　59
3.3.1　推理组件架构　59

3.3.2 学习组件架构 62
3.4 设计电子邮件模型 64
3.4.1 选择元素 64
3.4.2 定义依赖关系 67
3.4.3 定义函数形式 68
3.4.4 使用数值参数 71
3.4.5 使用辅助知识 73
3.5 构建推理组件 74
3.6 创建学习组件 78
3.7 小结 81
3.8 练习 82

第 2 部分 编写概率程序

第 4 章 概率模型和概率程序 85
4.1 概率模型定义 86
4.1.1 将一般知识表达为可能世界上的某种概率分布 86
4.1.2 进一步探索概率分布 88
4.2 使用概率模型回答查询 90
4.2.1 根据证据调节以产生后验概率分布 90
4.2.2 回答查询 92
4.2.3 使用概率推理 94
4.3 概率模型的组成部分 94
4.3.1 变量 95
4.3.2 依赖性 96
4.3.3 函数形式 101
4.3.4 数值参数 104
4.4 生成过程 105
4.5 使用连续变量的模型 110
4.5.1 使用β-二项式模型 110
4.5.2 连续变量的表示 111
4.6 小结 114
4.7 练习 114

第 5 章 用贝叶斯和马尔科夫网络建立依赖性模型 116
5.1 建立依赖性模型 117
5.1.1 有向依赖性 117
5.1.2 无向依赖性 122
5.1.3 直接和间接依赖性 124
5.2 使用贝叶斯网络 126
5.2.1 贝叶斯网络定义 126
5.2.2 贝叶斯网络如何定义概率分布 127
5.2.3 用贝叶斯网络进行推理 128
5.3 探索贝叶斯网络的一个示例 131
5.3.1 设计一个计算机系统诊断模型 131
5.3.2 用计算机系统诊断模型进行推理 135
5.4 使用概率编程扩展贝叶斯网络：预测产品的成功 140
5.4.1 设计产品成功预测模型 140
5.4.2 用产品成功预测模型进行推理 145
5.5 使用马尔科夫网络 147
5.5.1 马尔科夫网络定义 147
5.5.2 表示马尔科夫网络并用其进行推理 150
5.6 小结 153
5.7 练习 153

第 6 章 使用 Scala 和 Figaro 集合构建模型 156
6.1 使用 Scala 集合 157
6.1.1 为依赖于单一变量的多个变量建立模型 157
6.1.2 创建层次化模型 160
6.1.3 建立同时依赖两个变量的模型 162
6.2 使用 Figaro 集合 165
6.2.1 理解 Figaro 集合的用途 165
6.2.2 用 Figaro 集合重新实现层次化模型 166
6.2.3 结合使用 Scala 和 Figaro 集合 168

6.3 建立对象数量未知情况的模型 171
6.3.1 开放宇宙中对象数量未知的情况 171
6.3.2 可变大小数组 172
6.3.3 可变大小数组上的操作 172
6.3.4 示例：预测数量未知的新产品销售额 173
6.4 处理无限过程 174
6.4.1 Process 特征 175
6.4.2 示例：一个健康时空过程 176
6.4.3 使用过程 178
6.5 小结 179
6.6 练习 180

第 7 章 面向对象概率建模 182
7.1 使用面向对象概率模型 183
7.1.1 理解面向对象建模的元素 183
7.1.2 重温打印机模型 185
7.1.3 关于多台打印机的推理 189
7.2 用关系扩展 OO 概率模型 192
7.2.1 描述通用类级模型 192
7.2.2 描述某种情况 195
7.2.3 用 Figaro 表现社会化媒体模型 198
7.3 建立关系和类型不确定性的模型 200
7.3.1 元素集合和引用 200
7.3.2 具有关系不确定性的社会化媒体模型 202
7.3.3 具有类型不确定性的打印机模型 205
7.4 小结 207
7.5 练习 207

第 8 章 动态系统建模 209
8.1 动态概率模型 210
8.2 动态模型类型 211
8.2.1 马尔科夫链 211
8.2.2 隐含马尔科夫模型 214
8.2.3 动态贝叶斯网络 216
8.2.4 结构随时间改变的模型 220
8.3 建立永续系统的模型 224
8.3.1 理解 Figaro 的宇宙概念 224
8.3.2 使用宇宙建立持续运行系统的模型 225
8.3.3 运行一个监控应用 227
8.4 小结 229
8.5 练习 230

第 3 部分 推 理

第 9 章 概率推理三原则 235
9.1 链式法则：从条件概率分布构建联合分布 237
9.2 全概率公式：从联合分布获得简单查询结果 240
9.3 贝叶斯法则：从结果推断原因 243
9.3.1 理解、原因、结果和推理 243
9.3.2 实践中的贝叶斯法则 245
9.4 贝叶斯建模 247
9.4.1 估算硬币的偏差 248
9.4.2 预测下一次掷币结果 252
9.5 小结 256
9.6 练习 256

第 10 章 因子分解推理算法 258
10.1 因子 259
10.1.1 什么是因子 259
10.1.2 用链式法则分解概率分布 261
10.1.3 使用全概率公式，定义包含因子的查询 263

10.2 变量消除算法 267
10.2.1 VE 的图形解释 267
10.2.2 VE 代数运算 271
10.3 VE 的使用 273
10.3.1 Figaro 特有的 VE 考虑因素 273
10.3.2 设计模型支持高效的 VE 275
10.3.3 VE 的应用 278
10.4 置信传播 281
10.4.1 BP 基本原理 282
10.4.2 Loopy BP 的属性 282
10.5 BP 的使用 284
10.5.1 Figaro 特有的 BP 考虑因素 284
10.5.2 设计模型以支持高效的 BP 285
10.5.3 BP 的应用 287
10.6 小结 288
10.7 练习 288

第 11 章 抽样算法 291
11.1 抽样的原理 292
11.1.1 前向抽样 293
11.1.2 拒绝抽样 297
11.2 重要性抽样 299
11.2.1 重要性抽样的工作方式 300
11.2.2 在 Figaro 中使用重要性抽样 303
11.2.3 让重要性抽样为您工作 304
11.2.4 重要性抽样的应用 305
11.3 马尔科夫链蒙特卡洛抽样 307
11.3.1 MCMC 的工作方式 308
11.3.2 Figaro 的 MCMC 算法: Metropolis-Hastings 算法 311
11.4 让 MH 更好地工作 314
11.4.1 自定义提议 316
11.4.2 避免硬条件 319
11.4.3 MH 的应用 320
11.5 小结 321
11.6 练习 322

第 12 章 处理其他推理任务 324
12.1 计算联合分布 325
12.2 计算最可能的解释 326
12.2.1 在 Figaro 中计算和查询 MPE 329
12.2.2 MPE 查询算法的使用 331
12.2.3 探索 MPE 算法的应用 336
12.3 计算证据的概率 337
12.3.1 观察用于证据概率计算的证据 338
12.3.2 运行证据概率算法 339
12.4 小结 341
12.5 练习 341

第 13 章 动态推理和参数学习 342
13.1 监控动态系统的状态 342
13.1.1 监控机制 344
13.1.2 粒子过滤算法 345
13.1.3 过滤的应用 348
13.2 学习模型参数 349
13.2.1 贝叶斯学习 349
13.2.2 最大似然和 MAP 学习 353
13.3 进一步应用 Figaro 360
13.4 小结 361
13.5 练习 361

附录 A 获取和安装 Scala 和 Figaro 364
A.1 使用 sbt 364
A.2 在没有 sbt 的情况下安装和运行 Figaro 365
A.3 从源代码编译 366

附录 B 概率编程系统简况 367

第 1 部分

概率编程和 Figaro 简介

什么是概率编程？它有什么用处？如何使用它？这些问题是第 1 部分的主题。第 1 章介绍概率编程的基本思路。首先介绍概率推理系统的概念，说明概率编程如何将传统的概率推理系统概念和编程语言技术相结合。

在本书中，您将使用 Figaro 概率编程系统。第 1 章简要介绍 Figaro，第 2 章提供所有 Figaro 主要概念的简单教程，帮助您快速开始编写概率程序。第 3 章介绍一个完整的概率编程应用程序，为您提供实际应用程序组合的全貌。虽然这一章接近全书的开头，因此您从一开始就一窥全局，但是在阅读本书的更多章节，已经学习到更深入的概念时，仍值得不时复习。

第 1 章　概率编程简介

本章介绍如下内容：

- 什么是概率编程?
- 为什么应该关心概率编程？为什么我的老板应该关心概率编程?
- 概率编程的工作原理是什么?
- Figaro——概率编程所用的系统
- 使用和不使用概率编程的情况下，概率应用程序编写的对比

在本章中，您将学习如何使用概率推理系统的两个主要组成部分（概率模型和推理算法）做出日常决策，还将了解现代概率编程语言是如何比 Java 或 Python 等通用语言更轻松地创建这种推理系统的。本章还将介绍 **Figaro**，这是本书自始至终使用的基于 Scala 的概率编程语言。

1.1 什么是概率编程

概率编程是一种系统创建方法，它所创建的系统能够帮助我们在面对不确定性时做出决策。许多日常决策涉及在确定无法直接观测的相关因素时的判断能力。历史上，帮助在不确定性下做出决策的方法之一是使用概率推理系统。**概率推理**将我们对某种情况的认识和概率法则结合起来，确定无法观测的决策关键因素。直到最近，概率推理系统的范围仍然有限，难以应用到许多现实情况中。概率编程是一种新方法，它使概率推理系统更容易构建，适用范围更广。

要理解概率编程，首先要观察不确定性条件下的决策过程和涉及的主观判断。然后，您将了解概率推理是如何帮助您做出决策的。您将注意到概率推理系统所能进行的 3 种推理，也就能理解概率编程，以及通过编程语言的能力用概率编程构建概率推理系统的方法。

1.1.1 我们如何做出主观判断

在现实世界中，我们所关心的问题很少有非此即彼的答案。例如，如果您打算启动一个新产品，想要知道它的销路如何。您可能认为它将取得成功，因为您相信它设计精良，市场调查也表明有需求，但是无法确定。您的竞争者可能推出更好的产品，或者您的产品可能有市场不能容许的致命缺陷，经济也可能突然衰退。如果要求百分之百的确定，就无法做出是否投放该产品的决策（见图 1-1）。

概率语言有助于做出此类决策。在投放某个产品时，可以使用类似产品的先期经验估算产品的成功概率。然后，用这一概率帮助决定是否继续推进并投放该产品。您可能不仅关心产品能否成功，还关心它能带来多少收入，或者失败将导致多大的损失。您可以使用不同结果的概率做出更明智的决策。

概率论思想可以帮助您做出艰难的决策和判断，但是，您该怎么做呢？一般原则在下面列出。

事实：主观判断基于**知识+逻辑**。

您对感兴趣的问题有某些知识。例如，您对产品有深入的了解，可能进行了一些市场调查以找出客户的需求。您还可能有关于竞争对手的情报和经济预测。同时，逻辑帮助您运用知识获得问题的答案。



图 1-1　去年所有人都喜爱我的产品，但是明年会怎么样呢?

您需要一种规格化知识的方法，还需要运用知识得出问题答案的逻辑。概率编程提供了规格化知识和回答问题的逻辑。在我描述概率编程系统概念之前，我将描述概率推理系统的一般概念，这种系统提供了规格化知识和提供逻辑的基本手段。

1.1.2　概率推理系统帮助决策

概率推理是使用您的领域模型做出不确定条件下决策的一种方法。举个足球界的例子。假定统计显示 9%的角球造成进球。您的任务是预测某次角球的结果。攻方的中锋身高 6 英尺 4 英寸（约 1.93 米），以头球能力著称。守方正选门将刚刚受伤，被第一次出场的替补门将换下。除此之外，咆哮的大风使长传难以控制。那么，如何计算

进球的概率？

图 1-2 展示了使用概率推理系统找出答案的途径。您在一个角球模型中编码关于角球和所有相关因素的知识。然后，提供特定角球的证据，也就是中锋个子很高、守门员缺乏经验以及强风。您告诉该系统，希望知道这次角球是否进球。推理算法返回答案——有 20%的概率进球。

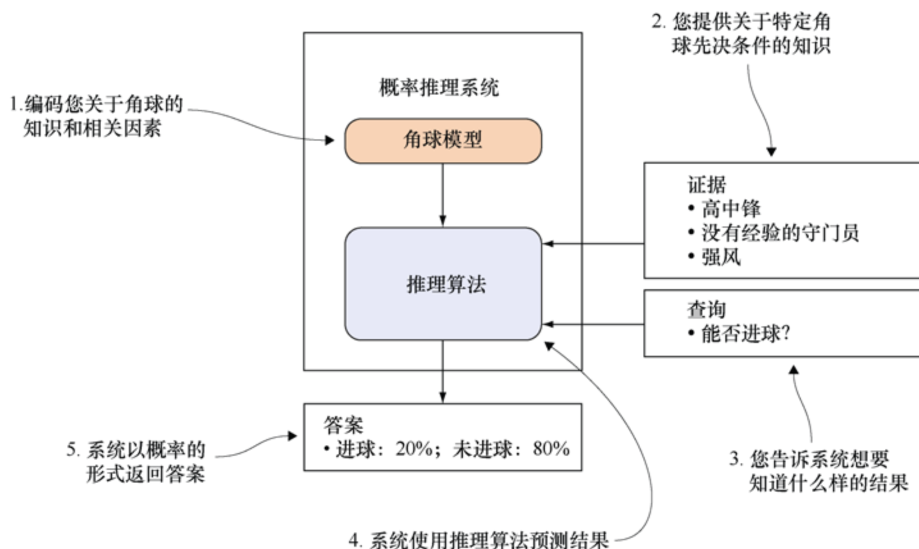


图 1-2 概率推理系统预测角球结果的方法

关键定义

一般知识——不考虑特定情况细节时，对领域相关情况的概括了解。

概率模型——用定量的概率术语编码的领域一般知识。

证据——关于特定情况的具体信息。

查询——您希望知道的情况属性。

推理——概率模型根据证据回答查询的过程。

在概率推理中，您创建一个**模型**，以定量的概率术语捕捉领域的所有相关一般知识。在我们的例子中，这个模型可能是对角球情况和影响结果的所有球员相关特征及条件的描述。然后，对于某个特定情况，您将该模型应用于所拥有的具体信息，得出结论。这些具体信息称为**证据**。在本例中，证据是中锋身材高大，守门员缺乏经验，风力很大。所得出的结论可以帮助您决策——例如，您是否应该在下一场比赛中更换不同的守门员。结论本身以概率的方式描述，比如守门员的不同技能水平的概率。

模型、您所提供的信息和查询答案之间的关系由数学上的概率法则定义。根据证据，运用模型回答查询的过程称作**概率推理**或者简单地称作**推理**。幸运的是，计算机算法已经有了很大的发展，能够为您完成这些数学题，自动进行所有必要的计算。这些算法被称作**推理算法**。

图 1-3 总结了您所学到的知识。

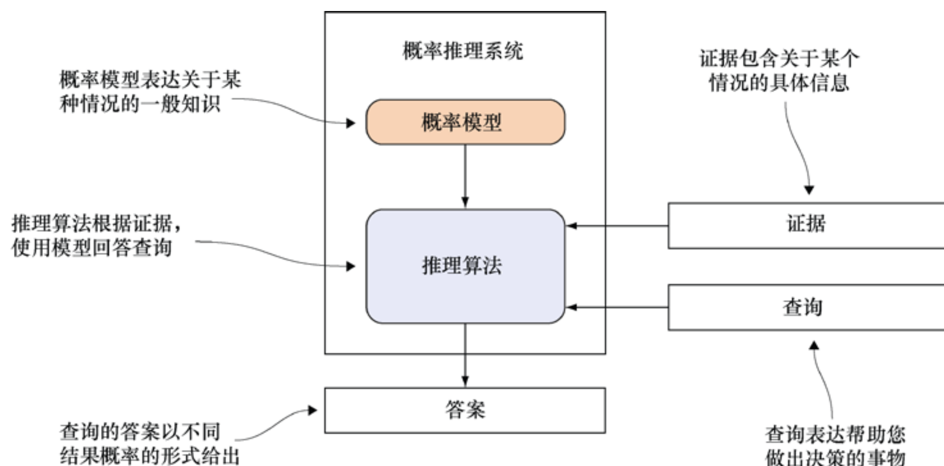


图 1-3　概率推理系统的基本组成部分

简言之，我们刚刚讨论的是概率推理系统的组成，以及与之互动的方式。但是，如何利用这样的系统？它如何帮助您决策？下一小节描述了概率推理系统所能执行的 3 类推理。

1.1.3　概率推理系统有 3 种方式推理

概率推理系统很灵活。它们可以根据任何方面的证据，回答关于情况其他特征的查询。在实践中，概率推理系统执行 3 类推理。

- **预测未来的事件**。在图 1-2 中您已经看到此类推理，根据当前情况预测是否进球。您的证据通常包括关于当前情况的信息，如中锋身高、守门员的经验和风力。
- **推断事件的根源**。快进 10 秒。高个中锋刚刚头球射门，从守门员身下入网，取得一分。根据这一证据，您对这位新手守门员有何想法？您能否得出结论，她的技能不足？图 1-4 说明如何使用概率推理系统回答这个问题。该模型是您之前用于预测是否进球的同一个角球模型（这是概率推理的一个实用属性：用于

预测未来结果的模型同样可以在事后推断结果的根源）。使用的证据和以前一样，并结合了角球得分这一事实。查询是守门员的技能水平，答案提供了不同技能水平的概率。

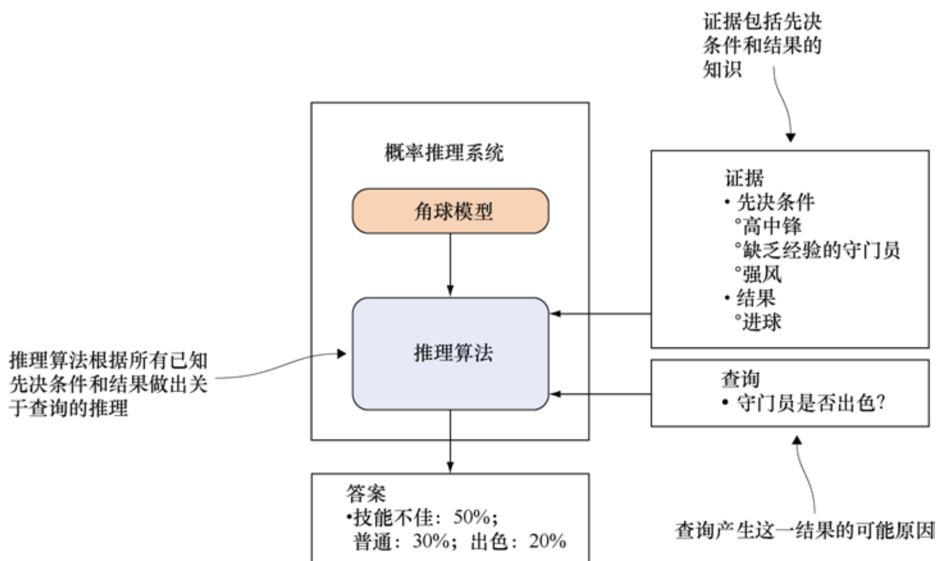


图 1-4 改变查询和证据，系统现在可以推断出进球的原因

想想看，第一种推理模式描述了前向推理，根据对当前情况的了解预测未来的事件，而第二种推理模式描述了后向推理，根据当前结果推断过去的条件。在构建概率模型时，模型本身通常遵循自然的时间顺序。一名球员踢角球，风作用于球，中锋跃起头球，守门员做出扑救。但是推理可以向前和向后进行。这是概率推理的关键特征之一，我在本书中将反复重申这一点：推理的方向不一定遵循模型的方向。

- **从过去的事件中学习，更好地预测未来的事件**。现在，再快进 10 分钟。同一球队又获得一次角球机会。所有情况与前面类似——高中锋、缺乏经验的守门员，但是现在风力减弱了。使用概率推理，可以利用前一次角球发生的情况，帮助您预测下一次角球的结果。图 1-5 说明了这一点。证据包括上一次的所有证据（注明其来自上一次）以及当前情况的新信息。在回答这次角球能否进球时，推理算法首先推断导致第一次进球的条件，例如中锋和守门员的技能水平。然后，它利用这些更新的属性做出关于新情况的决策。

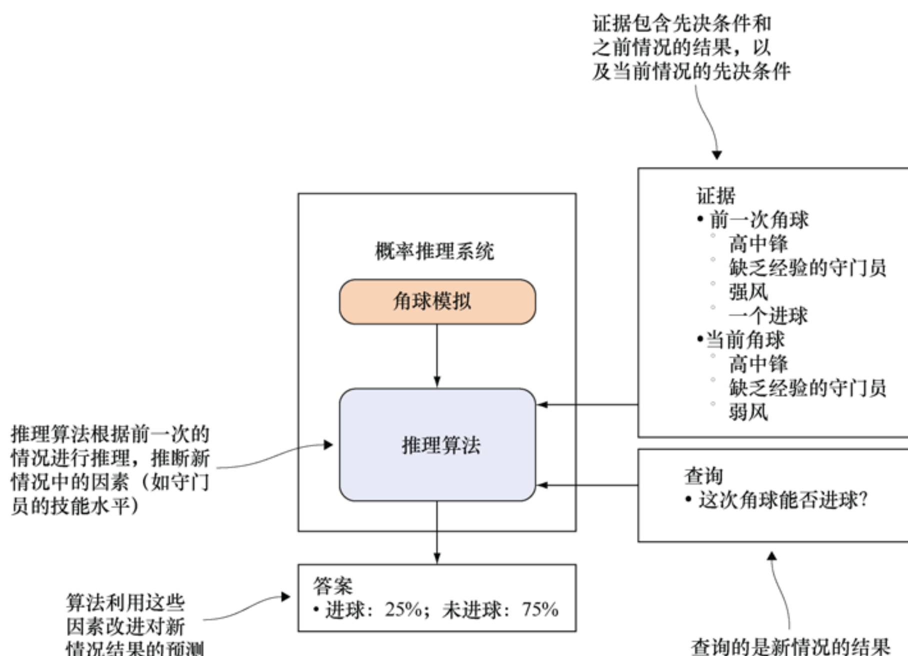


图 1-5 通过将上一次角球的结果考虑在内，概率推理系统可以在下一次角球时做出更好的预测

这些类型的查询能够帮助您做出许多层次上的决策。

- 您可以根据有无额外防守队员进球的概率，决定是否用一名防守队员替换进攻队员。
- 可以根据对守门员技能的评估，决定下一次合同谈判时向他提出的工资数额。
- 可以利用了解到的守门员相关情况，帮助预测下一场比赛的结果，决定是否使用同一名守门员。

学习更好的模型

上述 3 种推理模式提供了特定情况、给定证据下的推理手段，利用概率推理系统，还可以从过去的情况中学习，改善您的一般知识。在第三种推理模式中，您了解到如何从特定的过去经验学习，更好地预测未来的情况。另一种从过去的经验中学习的方法是改善模型本身。特别是在拥有许多过去的经验可以吸取时（如许多次角球），您可能希望学习一个新模型，以表示角球通常发生情况的一般知识。如图 1-6 所示，这可以通过一个学习算法实现。与推理算法有些不同，学习算法的目标是产生新的模型而不是回答查询。学习算法从原始模型入手，根据经验更新之，产生新的模型。新模型可以用于回答未来的问题。可以推测，使用新模型产生的答案应该比原始模型更明智。

概率推理系统与精确的预测

和任何机器学习系统一样，概率推理系统得到的数据越多，预测就越精确。预测的质量取决于两个因素：原始模型精确反映现实情况的程度和您所提供的数据量。一般来说，提供的数据越多，原始模型就越不重要，这是因为新模型是原始模型和数据所包含信息之间的一个平衡。如果您的数据很少，原始模型占据统治地位，所以它的质量必须很高才能得出准确的预测。如果您拥有许多数据，数据将占据统治地位，新模型倾向于忘掉不那么重要的原始模型。例如，如果您从整个足球赛季中学习，应该能够准确地学习到影响角球的因素。如果只有一场比赛的数据，就需要首先对精确预测比赛所需的因素有出色的想法。概率推理系统将很好地利用给定的模型和可用数据，尽可能精确地做出预测。

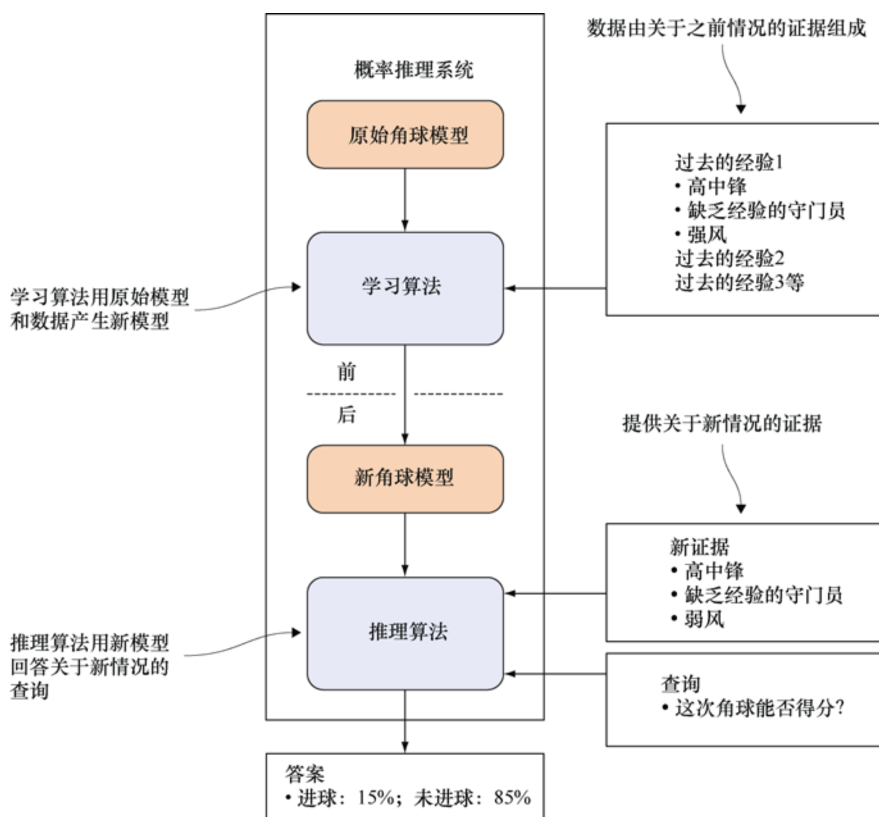


图 1-6 可以使用学习算法，以一组经验为基础学习新的模型。然后，这个新模型可以用于未来的推断

现在，您已经了解了概率推理的概念。那么，什么是概率编程？

1.1.4　概率编程系统：用编程语言表达的概率推理系统

每个概率推理系统都使用某种**表示语言**表达其概率模型。表示语言有许多种，您可能已经听说了其中一些，如贝叶斯网络（也称作置信网络）和隐含马尔科夫模型。表示语言控制系统可处理的模型以及模型的情况。语言所能表示的一组模型称作语言的**表达能力**。对于实际应用，您肯定希望表达能力尽可能强。

简单地说，**概率编程系统**是以编程语言作为表示语言的概率推理系统。我所说的**编程语言**是指具有编程语言所有预期特征（如变量、丰富的数据类型、控制流、函数等）的语言。正如您将要看到的，概率编程语言可以表达极其广泛的概率模型，超越传统的概率推理框架。概率编程语言有极强的表达能力。

图 1-7 说明了概率编程系统与概率推理系统的关系。可以将该图与图 1-3 比较，以凸显两种系统之间的差别。主要的变化是，模型以编程语言编写的程序表达，而不使用贝叶斯网络等数学结构。由于这种变化，证据、查询和答案都应用到程序中的变量。证据可能指定程序变量的特定值，查询询问程序变量的值，答案是不同查询变量值的概率。此外，概率编程系统通常带有一套推理算法。这些算法适用于以该语言编写的程序。

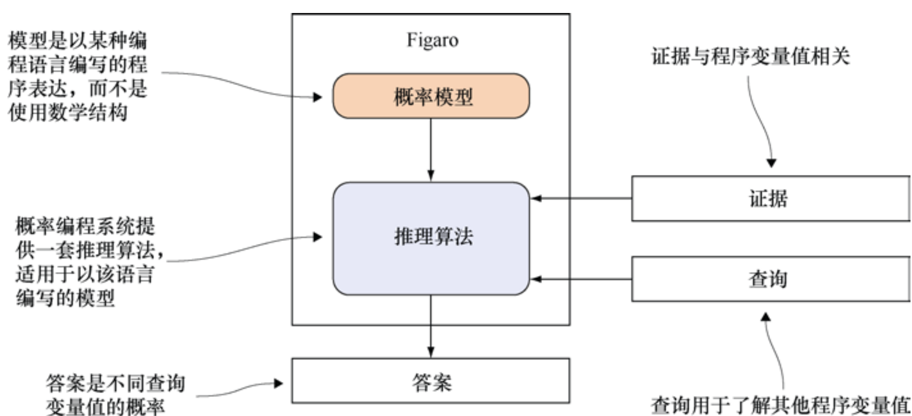


图 1-7　概率编程系统是使用编程语言表示概率模型的概率推理系统

尽管存在许多类概率编程系统（参见附录 B），本书的重点是函数式的图灵完备系统。**函数式**意味着它们基于函数式编程，但是不要被它吓住——使用函数式概率编程系统并不需要知道 λ 函数（lambda）等概念。这一切只意味着，函数式编程提供了这些语言表示概率模型的理论基础。同时，**图灵完备**是一句行话，表示编程语言可以编

写任何能在数字计算机上完成的计算。如果某一运算可以在数字计算机上完成，就可以由任何图灵完备语言实现。您所熟悉的大部分编程语言，如 C、Java 和 Python，都是图灵完备的。因为概率编程语言构建于图灵完备编程语言基础上，它们可以构建的模型类型极其灵活。

关键定义

表示语言——用于编码关于模型领域知识的语言。

表达能力——表示语言编码模型中不同类型知识的能力。

图灵完备——能够表示可在数字计算机完成的任何计算的语言。

概率编程语言——使用图灵完备编程语言表示知识的概率表示语言。

附录 B 论述了除本书使用的 Figaro 之外的一些概率编程系统。这些系统大部分都使用图灵完备语言。有一些系统（包括 BUGS 和 Dimple）没有使用图灵完备语言，但是它们对目标应用很实用。本书主要关注图灵完备概率编程语言的能力。

将概率模型表示为程序

但是，编程语言如何成为概率建模语言？如何将概率模型表示为程序？我将在这里提出回答这一问题的一些线索，将更深入的讨论放在稍后的章节，那时您已经对概率程序有所了解。

编程语言的核心思路之一是**执行**。您执行一个程序以产生输出。概率程序也类似，但是它可以有许多执行路径，每个路径产生不同的输出。在程序中随机选择执行路径，每个随机的选择有许多可能的结果，程序编码每种结果的概率。因此，概率程序可以视为随机执行以产生输出的一个程序。

图 1-8 说明了上述概念。在图中，概率编程系统包含了一个角球程序。这个程序描述生成角球结果的随机过程，它取得一些输入；在我们的例子中，这些输入是中锋的身高、守门员的经验和风力。根据这些输入，程序随机执行以生成输出。每次随机执行产生特定的输出。因为每个随机选择都有多种可能结果，存在许多可能的执行路径，造成不同的输出。任何给定输出（如进球）可能由多个执行路径产生。

让我们来看看，这种程序如何定义概率模型。从一系列随机选择形成的任何特定执行路径都有特定的结果。每个随机选择都有发生的概率。如果将这些概率相乘，就可以得到执行路径的概率。这样，程序定义了每个执行路径的概率。想象一下，如果将该程序运行许多次，生成任何给定执行路径的次数比例等于其概率。输出的概率就是产生该输出的程序运行次数比例。在图 1-8 中，1/4 的运行产生进球的结果，所以进球概率为 1/4。

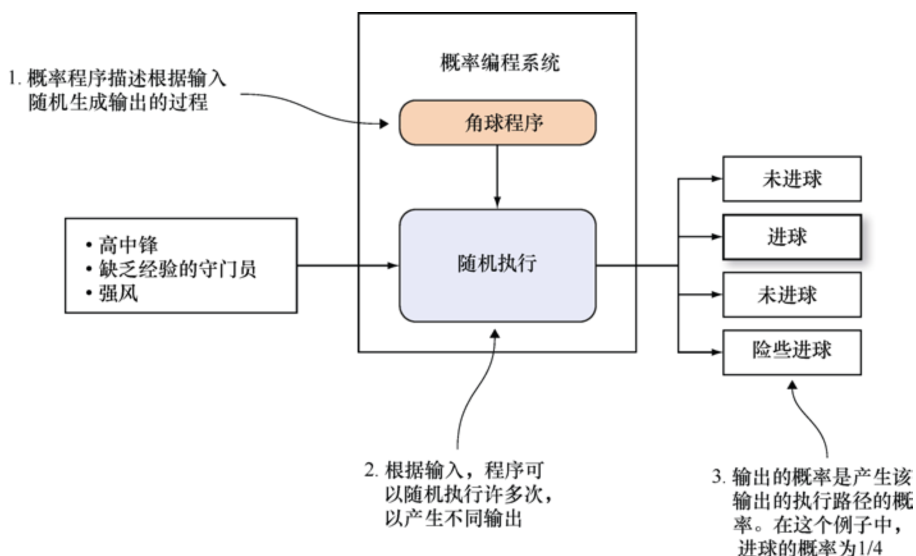


图 1-8　概率程序定义按照输入随机生成输出的过程

注意：您可能疑惑于为什么图 1-8 中的块标签为“随机执行”而不是其他插图中的推理算法。图 1-8 展示了概率程序的含义——定义一个随机执行过程，而不是使用概率编程系统的方式——使用推理算法根据证据回答查询。所以，尽管上述插图的结构类似，但是表达了不同的概念。事实上，随机执行形成了某些推理算法的基础，但是许多算法并不基于简单的随机执行。

利用概率编程决策

使用概率编程预测未来很容易理解。只要随机多次执行程序，使用当前已知的信息作为输入，并观察每个输出的出现次数。在图 1-8 的角球示例中，多次执行该程序，以高中锋、缺乏经验的守门员和强风作为输入。因为 1/4 的运行得出进球的结果，您可以认定在这些输入条件下，进球概率为 25%。

但是，概率编程的魔法在于，它还可以用于 1.3.1 小节中描述的各类概率推理。概率编程不仅可用于预测未来，还可以推断导致特定结果的事实；您可以“展开”程序，发现结果的根源，还可以在某种情况下应用程序，从结果中学习，在未来使用学习到的信息做出更好的决策。可以使用概率编程做出所有通过概率思想得到的决策。

概率编程是如何工作的？当人们意识到，在较简单的表示语言（如贝叶斯网络）上有效的推理算法可以扩展到程序上时，概率编程就变得实用了。本书的第 3 部分介绍实

现这一扩展的各种推理算法。幸运的是，概率编程系统自带一些内建的推理算法，这些算法可以自动地应用到您的程序中。您所需要做的是以概率程序的形式提供领域知识并指明证据，系统负责推断和学习。

在本书中，您将学习通过概率编程进行概率推理。首先，您将学习概率模型的概念以及使用它得出结论的方法。您还将学习一些从简单组件构成的模型中得出那些结论所需进行的操作。您将学习各种建模技术，以及使用概率编程实现它们的方法，还将了解概率推理算法的工作原理，以便有效地设计和使用自己的模型。在阅读完本书之后，您将能够自信地使用概率编程得出有益的结论，帮助您在面对不确定性时做出决策。

1.2 为什么使用概率编程

概率推理是机器学习的基础技术之一。Google、Amazon 和 Microsoft 等公司使用它理解可用数据。概率推理已经用于各种各样的应用程序，如预测股价、推荐电影、诊断计算机和检测网络入侵。许多应用都使用了本书中将要学习的技术。

前一小节中，有两个引人注目的要点。

- 概率推理可用于预测未来、推断过去，以及从过去的事实中学习更好地预测未来。
- 概率编程是使用图灵完备编程语言作为表示语言的概率推理。

将上面两个要点结合起来，可以得到如下表示。

事实：概率推理+图灵完备=概率编程

概率编程的动机是将两个本身就很强大的概念结合起来，结果是使用计算机辅助不确定性下决策的更简单、更灵活方法。

1.2.1 更好的概率推理

大部分现有概率表示语言在所能表示的系统丰富性上都很有限。有些相对简单的语言（如贝叶斯网络）假定固定的变量集，其灵活性不足，不能建立变量本身可能变化的领域模型。近年来，已经有一些具有更高灵活性的先进语言开发出来。其中一些语言（如 BUGS）还提供了编程语言的特征，包括循环和数组，但是没有达到图灵完备。BUGS 等语言的成功说明了更丰富、结构更严整的表示方式的必要性。但是，向成熟的图灵完备语言转移，为概率推理开拓了一个新领域。现在，可以建立具有许多交互实体及事件的长期运行过程的模型。

我们再次考虑足球的例子，但是这次想象一下，您的工作是体育分析，希望为一支球队做出人员配备决策的建议。您可以使用积累的统计数字做出决策，但是统计数字不能捕捉积累它们时所处的背景。您可以建立赛季的细致模型，实现粒度更细、情境感知

的分析。这要求建立许多相关事件以及相互作用的球员和球队的模型。如果没有完整的编程语言所提供的数据结构和控制流，构建这种模型是难以想象的。

现在，让我们再次思考产品投放的例子，从综合的角度观察业务决策过程。产品投放不是孤立事件，而是经过市场分析、研究和开发的过程，各个过程的结果都有不确定性。产品投放的结果取决于所有阶段，以及市场中其他产品的分析。全面的分析还需要关注竞争对手对您的产品的反应，以及他们可能提出的新产品。这一问题很困难，因为您必须对竞争产品做出推测。甚至有一些竞争对手尚不为人所知。在这个例子中，产品是复杂过程产生的数据结构。同样，用完整的编程语言创建模型很有益处。

不过，概率编程的好处之一是，可以使用更简单的概率推理框架。概率编程系统可以表示广泛的现有框架，以及这些框架所不能表示的系统。本书将传授许多使用概率编程的此类框架。所以，在概率编程的学习中，您还能够精通许多当今常用的概率推理框架。

1.2.2　更好的模拟语言

图灵完备的概率建模语言已经存在。它们常常被称作**模拟语言**。我们知道，使用编程语言模拟足球赛季等复杂过程是可能的。在这种情境下，我使用**模拟语言**这一术语描述能够表示复杂过程随机执行的语言。正如概率程序，这些模拟随机执行，以产生不同输出。模拟和概率推理一样应用广泛，涵盖了从军事计划到组件设计以及公共卫生及体育比赛预测等范围。确实，精密模拟的广泛使用说明了对丰富概率建模语言的需求。

但是，概率程序远不仅是模拟。使用模拟，您只能完成概率程序的一项功能：预测未来。无法用它推断观测结果的根源。而且，尽管可以不断地用已知的当前信息更新模拟，但是很难包含必须推断的未知信息。因此，从过去经验中学习以改善未来预测和分析的能力很有限。不能将模拟用于机器学习。

概率程序就像不仅可以运行，而且可以分析的模拟一样。开发概率编程的关键要点是，推理算法既可用于较简单的建模框架，也可用于模拟。因此，您有能力编写一个模拟并在其基础上执行推理，以创建概率模型。

最后一点，概率推理系统已经出现了一段时间，Hugin、Netica 和 BayesiaLab 等软件提供了贝叶斯网络系统。但是概率编程更有表现力的表示语言很新颖，我们刚刚开始发现其强大的应用。老实说，我不能告诉您概率编程已经用于大量现有应用，但是有一些重要的应用。Microsoft 已经能够使用概率编程，确定在线游戏玩家的真正技能水平。加州大学伯克利分校的 Stuart Russell 编写了一个程序，通过识别表明核爆炸的地震活动，帮助联合国《全面禁止核试验条约》的实施。麻省理工学院（MIT）的 Josh Tenenbaum 和斯坦福大学的 Noah Goodman 已经创建了建立人类识别模型的概率

程序，并在试验中取得了很大的成功。在 Charles River Analytics，我们已经使用概率编程推断恶意软件实例的组件并确定它们的演变。但是，我相信这些应用仅仅是个开始。将会有越来越多的人用概率编程系统做出所在领域的决策。阅读本书，您也有机会成为这一新技术的尝鲜者。

1.3 Figaro 简介：一种概率编程语言

在本书中，您将使用一种称为 Figaro 的概率编程系统。（我用莫扎特的歌剧《费加罗的婚礼》中的角色为其命名。我喜爱莫扎特，并在该剧于波士顿的一次演出中饰演巴尔托洛医生。）本书的主要目标是教授概率编程的原则，在本书中学到的技术应该可以在其他概率编程系统上沿用。附录 B 简单描述了现有的一些系统。但是，本书还有第二个目标——帮助您获得创建使用概率程序的亲身体验，并提供可以立即使用的工具。因此，许多例子都用 Figaro 代码实现。

Figaro 是从 2009 年开始开发的一个开源软件，在 GitHub 上维护。它以 Scala 库的形式实现。图 1-9 说明 Figaro 如何使用 Scala 实现概率编程系统。该图详细说明了图 1-7，

图 1-9 Figaro 使用 Scala 提供概率编程系统的方法

后者描述了概率编程系统的主要组成部分。让我们从概率模型开始，在 Figaro 中，该模型由任意数量的数据结构（称作“**元素**”）组成。每个元素代表在您的情境中可取任意数量值的一个变量。这些数据结构用 Scala 实现，您可以用这些数据结构编写 Scala 程序创建模型。可以通过关于元素值的信息提供证据，也可以指定希望在查询中了解的元素。至于推理算法，您可以选择一个 Figaro 内建推理算法并应用到模型上，根据证据回答您的查询。推理算法以 Scala 实现，其调用就是一个 Scala 函数调用。推理结果是查询元素不同值的概率。

Figaro 内嵌于 Scala 提供了一些重大优势。其中一些来自内嵌于通用宿主语言相对于独立概率语言的优势。其他优势则是因为 Scala 的良好特性。下面是在通用宿主语言中内嵌概率编程语言的好处。

- 证据可以用宿主语言的程序得出。例如，您可以编写一个程序读取一个数据文件，以某种方式处理其中的值，并将其作为证据提供给 Figaro 模型。在独立语言中，这一任务要难得多。
- 类似地，您可以在一个程序中使用 Figaro 提供的答案。例如，如果您有一个供足球队经理使用的程序，该程序可以取得进球概率，向经理提出建议。
- 可以在概率程序中嵌入通用代码。例如，假设您有一个模拟头球在空中飞行轨迹的物理模型，可以在 Figaro 元素中加入这个模型。
- 可以使用通用编程技术构建 Figaro 模型。例如，您可能有一个映射，包含对应于球队中所有球员的 Figaro 元素，并根据情况中涉及的球员选择对应的元素。

下面是选择 Scala 作为内嵌概率编程系统的宿主语言的一些理由。

- Scala 是一个函数式编程语言，因此 Figaro 也能得到函数式编程的好处。正如我在第 2 部分中所说明的，函数式编程对概率编程有帮助，许多模型可以自然地以函数式风格编写。
- Scala 是面向对象的，其优点之一是既是函数式语言，又具有面向对象的特征。Figaro 也是面向对象的。正如第 2 部分中将要说明的，面向对象是表达概率编程中多种设计模式的有用手段。

最后，Figaro 还有嵌入 Scala 之外的一些优势，包括：

- Figaro 能够表示极其广泛的概率模型。Figaro 元素的值可以为任何类型，包括布尔型、整数、双精度数、数组、树、图等。这些元素之间的关系可以由任何函数定义。
- Figaro 提供了使用其条件和约束规定证据的丰富框架。
- Figaro 有多种多样的推理算法。
- Figaro 能够表示和推理随时间变化的动态模型。
- Figaro 能够在其模型中包含明确决策，并支持最优决策的推断。

由于多种原因，Figaro 是学习概率编程的出色语言。

- Figaro 以 Scala 库的形式实现，可以用于 Java 和 Scala 程序，很容易与应用程序集成。
- 由于以程序库而非独立语言的形式实现，Figaro 提供了宿主编程语言的全部功能，可以用来构建模型。Scala 是高级的现代化语言，具有许多有用的程序组织功能，使用 Figaro 时可以自动获得这些好处。
- 从所提供算法的范围来看，Figaro 堪称全能。

本书强调使用的技术和实用的示例。只要有可能，我都会解释建模的一般原则，并描述在 Figaro 中的实现方法。不管您最终使用哪一种概率编程系统，这对您都将大有裨益。并不是所有系统都能轻松地实现本书中的所有技术。例如，现有的面向对象概率编程系统很少。但是有了好的基础，您就可以找出用所选语言表达需求的方法。

使用 Scala

因为 Figaro 是一个 Scala 库，需要 Scala 的知识才能使用 Figaro。本书是关于概率编程的，所以在本书中不教授 Scala 的知识。Scala 的出色学习资源很多，比如 Twitter 的 Scala School（http://twitter.github.io/scala_school）。但是为了防止您对 Scala 不自信，我在本书中对代码中使用的 Scala 功能加以说明。即使您还不了解 Scala，也能够跟上本书的进度。

从概率编程和 Figaro 中获益并不要求您是一位 Scala 奇才，在本书中也避免使用一些较为高级和晦涩的特性。但是，增强 Scala 技能有助于成为更好的 Figaro 程序员。您甚至会发现，阅读本书也可以提高 Scala 的技能。

Figaro 与 Java 的对比：构建简单的概率编程系统

为了说明概率编程和 Figaro 的好处，我将展示以两种方式编写的简单概率应用。首先，我说明用 Java（您可能对它很熟悉）编写这种应用的方法。然后，我将展示用 Figaro 编写的 Scala 应用。尽管 Scala 相对 Java 有一定的优势，但是这不是我要指出的主要差别。关键的思路是，Figaro **提供了表示概率模型和用这些模型进行推理的能力，如果没有概率编程，这些能力就不存在。**

我们的小应用将作为 Figaro 的“Hello，World”示例。想象一下，有个人早上起床，查看天气是否晴朗，并根据天气发出问候。每天发出连续两天的问候。而且，第二天的天气取决于第一天：如果第一天是晴天，第二天就更可能是晴天。这些陈述可以由表 1-1 中的数字量化。

表 1-1　量化“你好，世界”示例的概率

今天的天气		
晴天	0.2	
不是晴天	0.8	
今天的问候语		
如果今天是晴天	“Hello, world!”	0.6
	“Howdy, universe!”	0.4
如果今天不是晴天	“Hello, world!”	0.2
	“ Oh no, not again”	0.8
明天的天气		
如果今天是晴天	晴天	0.8
	不是晴天	0.2
如果今天不是晴天	晴天	0.05
	不是晴天	0.95
明天的问候语		
如果明天是晴天	“Hello, world!”	0.6
	“Howdy, universe!”	0.4
如果明天不是晴天	“Hello, world!”	0.2
	“Oh no, not again”	0.8

下面几章将明确解释这些数字的含义。现在，我们直观地认为今天是晴天的概率为 0.2，也就是说，今天有 20%的可能放晴。同样，如果明天是晴天，明天的问候语为“Hello, world!”的概率为 0.6，也就是说问候语为“Hello, world!”有 60%的可能性，“Howdy, universe!”的可能性为 40%。

我们为自己设定了用这个模型执行 3 种推理任务的目标。在 1.1.3 小节中您已经知道，用概率模型能够进行 3 类推理：**预测**未来，**推断**导致观测结果的过去事件，从过去事件中**学习**以更好地预测未来。您将用我们的简单模型完成这三种任务。具体任务如下。

1．预测今天的问候语。

2．如果观测发现今天的问候语是“Hello, world!”，推断今天是不是晴天。

3．从今天对问候语是“Hello, world!”这一观测值的学习，预测明天的问候语。

下面是用 Java 完成这些任务的方法。

程序清单 1-1 用 Java 实现的 Hello World 程序

```
class HelloWorldJava {
  static String greeting1 = "Hello, world!";
  static String greeting2 = "Howdy, universe!";
  static String greeting3 = "Oh no, not again";

  static Double pSunnyToday = 0.2;
  static Double pNotSunnyToday = 0.8;
  static Double pSunnyTomorrowIfSunnyToday = 0.8;
  static Double pNotSunnyTomorrowIfSunnyToday = 0.2;
  static Double pSunnyTomorrowIfNotSunnyToday = 0.05;
  static Double pNotSunnyTomorrowIfNotSunnyToday = 0.95;
  static Double pGreeting1TodayIfSunnyToday = 0.6;
  static Double pGreeting2TodayIfSunnyToday = 0.4;
  static Double pGreeting1TodayIfNotSunnyToday = 0.2;
  static Double pGreeting3TodayIfNotSunnyToday = 0.8;
  static Double pGreeting1TomorrowIfSunnyTomorrow = 0.6;
  static Double pGreeting2TomorrowIfSunnyTomorrow = 0.4;
  static Double pGreeting1TomorrowIfNotSunnyTomorrow = 0.2;
  static Double pGreeting3TomorrowIfNotSunnyTomorrow = 0.8;

  static void predict() {
     Double pGreeting1Today =
            pSunnyToday * pGreeting1TodayIfSunnyToday +
            pNotSunnyToday * pGreeting1TodayIfNotSunnyToday;
     System.out.println("Today's greeting is " + greeting1 +
        "with probability " + pGreeting1Today + ".");
  }

    static void infer() {
       Double pSunnyTodayAndGreeting1Today =
              pSunnyToday * pGreeting1TodayIfSunnyToday;
       Double pNotSunnyTodayAndGreeting1Today =
              pNotSunnyToday * pGreeting1TodayIfNotSunnyToday;
       Double pSunnyTodayGivenGreeting1Today =
              pSunnyTodayAndGreeting1Today /
               (pSunnyTodayAndGreeting1Today +
              pNotSunnyTodayAndGreeting1Today);
       System.out.println("If today's greeting is " + greeting1 +
           ", today's weather is sunny with probability " +
           pSunnyTodayGivenGreeting1Today + ".");
    }
```

定义问候语

指定模型的数值参数

用概率推理规则预测今天的问候语

按照今天的问候语是“Hello, world!”这一观测值，运用概率推理原则推断今天的天气

```
  static void learnAndPredict() {
     Double pSunnyTodayAndGreeting1Today =
            pSunnyToday * pGreeting1TodayIfSunnyToday;
     Double pNotSunnyTodayAndGreeting1Today =
            pNotSunnyToday * pGreeting1TodayIfNotSunnyToday;
     Double pSunnyTodayGivenGreeting1Today =
            pSunnyTodayAndGreeting1Today /
             (pSunnyTodayAndGreeting1Today +
                  pNotSunnyTodayAndGreeting1Today);
     Double pNotSunnyTodayGivenGreeting1Today =
            1 - pSunnyTodayGivenGreeting1Today;
     Double pSunnyTomorrowGivenGreeting1Today =
            pSunnyTodayGivenGreeting1Today *
                 pSunnyTomorrowIfSunnyToday +
            pNotSunnyTodayGivenGreeting1Today *
                 pSunnyTomorrowIfNotSunnyToday;
     Double pNotSunnyTomorrowGivenGreeting1Today =
            1 - pSunnyTomorrowGivenGreeting1Today;
     Double pGreeting1TomorrowGivenGreeting1Today =
            pSunnyTomorrowGivenGreeting1Today *
                 pGreeting1TomorrowIfSunnyTomorrow +
            pNotSunnyTomorrowGivenGreeting1Today *
                 pGreeting1TomorrowIfNotSunnyTomorrow;
     System.out.println("If today's greeting is " + greeting1 +
         ", tomorrow's greeting will be " + greeting1 +
         " with probability " +
         pGreeting1TomorrowGivenGreeting1Today);
  }

  public static void main(String[] args) {
    predict();
    infer();
    learnAndPredict();
  }
}
```

从今天问候语是“Hello, world!”的观测中学习，运用概率推理原则预测明天的问候语

执行所有任务的主方法

在此，我不对使用推理规则进行计算的方法做出描述。上述代码使用了 3 条推理规则：链式法则、全概率公式和贝叶斯法则。这些规则将在第 9 章中详细解释。现在，我们指出这段代码的两个主要问题。

■　**无法定义建模所用的规则**

模型定义包含在一个变量名与双精度值的列表中。当我在本节的开始描述模型，在表 1-1 中展示数值时，模型有许多结构，尽管不算很直观，但也算相对容易理解。变量

定义的列表毫无结构。变量的含义埋藏在变量名之中，这一定不是好主意。因此，以这种方式记下模型很难，该过程也很容易出错。以后阅读理解和维护这些代码也很困难。如果需要修改模型（例如，问候语还取决于您睡得好不好），就可能需要重写模型的很大一部分。

■ **自行编码推理规则很难且容易出错**

上述代码的第二个主要问题是使用概率推理规则回答查询。您必须有关于推理规则的详细知识才能编写这段代码。即使有了这种知识，正确编写代码也很难。测试答案是否正确也很困难，而这只是一个极其简单的例子。对于复杂的应用，以此方式创建推理代码可能不现实。

下面看看 Scala/Figaro 代码。

程序清单 1-2 用 Figaro 实现的 Hello World 程序

```
import com.cra.figaro.language.{Flip, Select}
import com.cra.figaro.library.compound.If
import com.cra.figaro.algorithm.factored.VariableElimination

object HelloWorld {
  val sunnyToday = Flip(0.2)
  val greetingToday = If(sunnyToday,
       Select(0.6 -> "Hello, world!", 0.4 -> "Howdy, universe!"),
       Select(0.2 -> "Hello, world!", 0.8 -> "Oh no, not again"))
  val sunnyTomorrow = If(sunnyToday, Flip(0.8), Flip(0.05))
  val greetingTomorrow = If(sunnyTomorrow,
       Select(0.6 -> "Hello, world!", 0.4 -> "Howdy, universe!"),
       Select(0.2 -> "Hello, world!", 0.8 -> "Oh no, not again"))

  def predict() {
    val result = VariableElimination.probability(greetingToday,
               "Hello, world!")
    println("Today’s greeting is \"Hello, world!\" " +
           "with probability " + result + ".")
  }

    def infer() {
      greetingToday.observe("Hello, world!")
      val result = VariableElimination.probability(sunnyToday, true)
      println("If today's greeting is \"Hello, world!\", today’s " +
             "weather is sunny with probability " + result + ".")
    }
```

导入 Figaro 结构

定义模型

用推理算法预测今天的问候语

根据今天的问候语是“Hello, world!”这一事实，使用推理算法推理今天的天气

```
  def learnAndPredict() {
    greetingToday.observe("Hello, world!")
    val result = VariableElimination.probability(greetingTomorrow,
                   "Hello, world!")
    println("If today's greeting is \"Hello, world!\", " +
            "tomorrow's greeting will be \"Hello, world!\" " +
            "with probability " + result + ".")
  }

  def main(args: Array[String]) {
    predict()
    infer()
    learnAndPredict()
  }
}
```

从对今天的问候语是“Hello，world!”这一观察中学习，用推理算法预测明天的问候语

执行所有任务的主方法

我将等到下一章才详细解释这段代码。现在，我希望指出，它解决了 Java 代码的两个问题。首先，模型定义准确描述了对应于表 1-1 的模型结构。您定义了 4 个变量：sunnyToday、greetingToday、sunnyTomorrow 和 greetingTomorrow，它们都对应于表 1-1。例如，greetingToday 的定义如下：

```
val greetingToday = If(sunnyToday,
       Select(0.6 -> "Hello, world!", 0.4 -> "Howdy, universe!"),
       Select(0.2 -> "Hello, world!", 0.8 -> "Oh no, not again"))
```

这段代码说明，如果今天是晴天，则问候语为“Hello，world!”的概率为 0.6，“Howdy, universe!”的概率是 0.4。如果今天不是晴天，问候语为“Hello，world!”的概率为 0.2，“Oh no, not again”的概率为 0.8。这正是表 1-1 对今天问候语的规定。因为代码明确地描述了模型，构造、阅读和维护就容易得多了。如果需要更改模型（例如，添加 sleepQuality 变量），可以用模块化的方式完成。

现在，我们来看看执行推理任务的代码，它没有包含任何计算，而是实例化一个算法（在本例中是一个变量消除算法，Figaro 中可用的几个算法之一），并查询该算法以获得所需的概率。现在，按照第 3 部分中的说明，这个算法基于和 Java 程序相同的概率推理规则。组织和应用推理规则的艰苦工作由该算法负责。即使对于大而复杂的模型，也可以运行该算法，完成所有推理。

1.4　小结

- 做出判断需要知识+逻辑。
- 在概率推理中，概率模型表示知识，推理算法编码逻辑。
- 概率推理可用于预测未来事件，推断过去事件的原因，从过去事件中学习以改

善预测。

- 概率编程是概率推理，其中的概率模型用编程语言表达。
- 概率编程系统使用图灵完备编程语言表示模型，并提供使用模型的推理算法。
- Figaro 是用 Scala 实现的概率编程系统，Scala 提供了函数和面向对象编程风格。

1.5　练习

部分练习的解答可在 www.manning.com/books/practical-probabilistic-programming 上找到。

1．想象一下，您打算用一个概率推理系统推理扑克牌型的结果。

a）您可以在模型中编码哪种一般知识？

b）描述如何使用系统预测未来。什么是证据？什么是查询？

c）描述如何使用系统推断当前观测结果的根源。什么是证据？什么是查询？

d）描述推断出的过去根源如何帮助您预测未来。

2．在 Hello World 示例中，根据如下表格改变今天天气是否晴朗的概率。程序输出有何变化？为什么您认为将出现这样的变化？

今天的天气	
晴朗	0.9
不晴朗	0.1

3．修改 Hello World 示例，添加一个新问候：“Hi，galaxy!”。提供这个问候语在天气晴朗时的概率，降低其他问候语的概率使总概率保持为 1。还要修改程序，使所有查询打印“Hi，galaxy!”的概率而不是“Hello, world!”的概率。用 Java 和 Figaro 版本的 Hello World 程序进行这一修改。比较两种语言的过程。

第 2 章　Figaro 快速教程

本章介绍如下内容：

- 模型创建、证据陈述、推理运行和查询的回答
- 理解模型基本构件
- 从这些构件构造复杂模型

现在，您已经了解了概率编程的含义，可以详细了解 Figaro，以便编写自己的简单程序，用它们回答查询。本章的目标是尽快介绍 Figaro 的最重要概念。后面的章节详细解释模型的含义以及对它们的理解。让我们开始吧。

2.1　Figaro 简介

首先，我们对 Figaro 做一个概述。在第 1 章中已经介绍过，Figaro 是一种概率推理系统。在查看其组件之前，我们先回顾概率推理系统的一般组件，使您可以和 Figaro 比较。图 2-1 重现了第 1 章中介绍的概率推理系统要点。提醒一下，关于情况的一般知识在概率模型中编码，而证据提供关于特定情况的具体信息。推理算法使用模型和证据回答关于情况的查询。

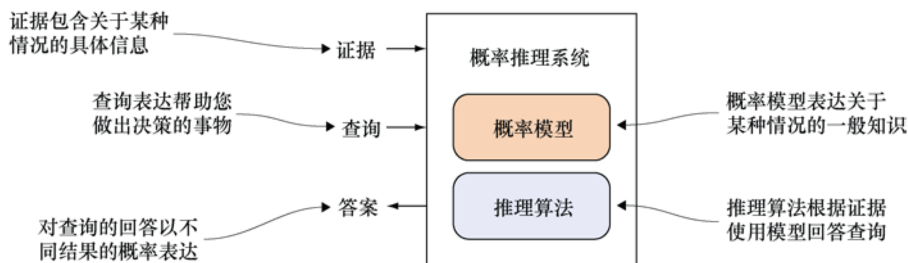


图 2-1 概率推理要点回顾

现在我们来看看 Figaro。图 2-2 展示了 Figaro 的关键概念。可以看到，该图和图 2-1 有着相同的组件。Figaro **模型**用来表达一般知识。您以**证据**的形式提供关于某种情况的具体知识。**查询**告诉系统您所感兴趣的发现。Figaro **推理算法**取得证据并用模型提供查询的**答案**。

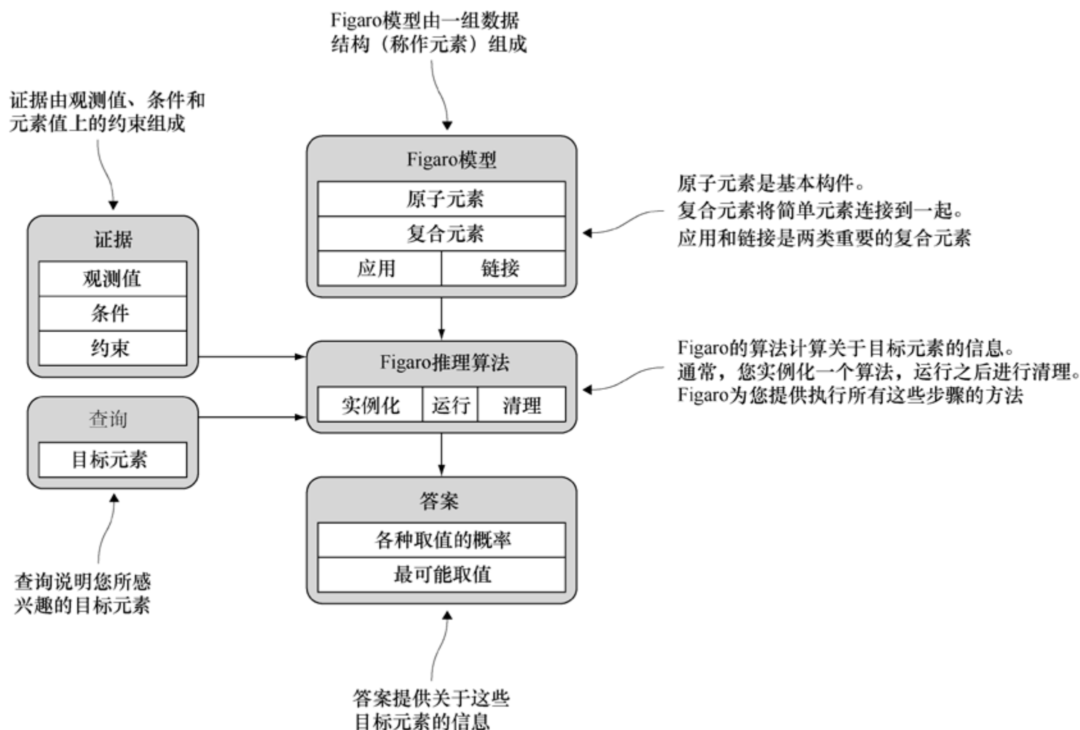


图 2-2 Figaro 的关键概念及其组合

现在，我们来观察每一个组件。Figaro 的大部分接口提供了指定 Figaro 模型的方法。Figaro 模型由一组称为**元素**的数据结构组成。每个元素代表在您的情况中可以取一组值之一的变量。元素编码定义不同值概率的信息。您将在 2.2.1 小节中 Hello World 示例的上下文中看到元素的基本定义。

元素主要有两类：原子元素和复合元素。您可以将 Figaro 视为构建模型的构造工具箱。**原子元素**是基本构件，表示不依赖于其他元素的基本概率变量。2.3 小节讨论原子元素并提供各种示例。**复合元素**是连接器，它们依赖于一个或者多个元素以构成更复杂的元素。您将在 2.4 小节中学习关于复合元素的知识。Figaro 提供种种不同的复合元素，其中两种特别重要——**Apply**（**应用**）和 **Chain**（**链**），您将在 2.5 小节中学习如何使用它们。

接下来的是证据。Figaro 提供了说明证据的丰富机制。大部分时候，您将使用证据的最简单形式——**观测值**。观测值指定已知有某个特定值的元素。您将在 2.2.3 小节学习如何指定观测值。有时候，您需要更通用的证据说明方法。为此 Figaro 提供了**条件**和**约束**。条件和约束及其用法在 2.6 小节中描述。

Figaro 的查询通过指明目标元素和您想知道的有关情况指明。您使用某种算法，按照证据找出有关目标元素的信息。通常，您必须**实例化**某种算法，**运行**并在之后**清理**。我已经提供了使用默认设置执行所有步骤的简单方法。在运行算法之后，您可以获得查询的答案。这些算法最常采取目标元素的**各种取值概率**。有时候，它们没有告诉您概率，而是得出每个目标元素的**最可能取值**。对于每个目标元素，答案告诉您最高概率的取值。您将在 2.2.2 小节中看到如何指定查询、运行算法和获得答案。

2.2 创建模型和运行推理：重回 Hello World

您已经概要了解了 Figaro 概念，接下来看看它们是如何融合在一起的。您将回顾第 1 章的 Hello World 示例，特别注意图 2-2 中的所有概念是如何出现在这个例子中的。您将关注如何从原子和复合元素中构建模型，观测证据，提出查询，运行推理算法，得到答案。

本章的代码可以两种方式运行。一种是使用 Scala 控制台，逐行输入语句并获得即时响应。为此，进入本书项目根目录 PracticalProbProg/examples 并输入 sbt console，将会看到 Scala 提示符。然后输入每行代码，查看响应。

第二种方式是通常的方法：编写一个包含 main 方法的程序，该方法包含想要执行的代码。在本章中，我不提供将代码转换为可运行程序的模板，只提供与 Figaro 相关的代码。我将确保指出您需要导入的内容及将其导入的位置。

2.2.1 构建第一个模型

首先，您将构建最简单的 Figaro 模型。这个模型包含一个原子元素。构建模型之前，必须导入必要的 Figaro 结构：

```
import com.cra.figaro.language._
```

上述语句导入 com.cra.figaro.language 包中的所有类，该包包含最基本的 Figaro 结构。这些类中有一个称为 Flip。可以用 Flip 构建一个简单模型：

```
val sunnyToday = Flip(0.2)
```

图 2-3 解释了这一行代码。搞清楚哪一部分是 Scala，哪一部分是 Figaro，是很重要的。在这行代码中，创建了一个名为 sunnyToday 的变量，并赋值 Flip(0.2)。Scala 值 Flip(0.2) 是一个 Figaro 元素，表示 true 值概率为 0.2、false 值概率为 0.8 的一个随机过程。**元素**是表示随机产生一个值的过程的数据结构。随机过程可能产生任意数量的结果。每个可能结果被称为过程的一个**值**。因此，Flip(0.2)是可能取值为布尔值 true 及 false 的元素。总结起来就是，您有了一个包含 Scala 值的 Scala 变量。该 Scala 值是 Figaro 元素，它包含表示过程不同结果的任意个可能取值。

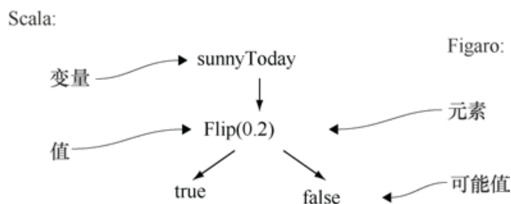


图 2-3 Scala 变量与值以及 Figaro 元素与可能值之间的关系

在 Scala 中，类型可以由另外一种描述其内容的类型参数化。您可能从 Java 泛型中已经熟悉了这个概念，例如，在 Java 中可以得到一个整数或者字符串的列表。所有 Figaro 元素都是 Element 类的实例。Element 类由元素可能取值的类型参数化。这种类型称作元素的**值类型**。因为 Flip(0.2)可以取布尔值，Flip(0.2)的值类型为 Boolean。这一事实的标记方法是：Flip(0.2)是 Element[Boolean]的一个实例。

关键定义

元素——代表一个随机过程的 Figaro 数据结构。
值——随机过程的一个可能结果。
值类型——代表元素可能取值的 Scala 类型。

关于这个简单模型有许多值得说明的地方。幸运的是，您已经学到的知识适用于所有

Figaro 模型。Figaro 模型通过取得和组合简单的 Figaro 元素（构件）创建更复杂的元素和相关元素集合而创建。您刚刚学到的元素、值和值类型的定义是 Figaro 中最为重要的定义。

在继续构建更复杂的模型之前，我们先来看看如何用这个简单模型进行推理。

2.2.2 运行推理和回答查询

您已经构建了一个简单模型。我们运行推理，查询 sunnyToday 为 true 的概率。首先，需要导入将要使用的推理算法：

```
import com.cra.figaro.algorithm.factored.VariableElimination
```

上述语句导入所谓的**“变量消除”（variable elimination）**算法，这是一种精确的推理算法，也就是说，它可以准确地计算您的模型和证据隐含的概率。概率推理很复杂，所以精确的算法有时候需要花费很长时间，或者耗尽内存。Figaro 提供近似算法，这种算法通常提供与准确答案大致相同的答案。本章使用简单的模型，所以变量消除算法在大部分情况下可行。

现在，Figaro 提供一个简单命令以指定查询、运行算法并获得答案。可以编写如下的代码：

```
println(VariableElimination.probability(sunnyToday, true))
```

上述命令打印输出 0.2。您的模型只包含元素 Flip(0.2)，结果为 true 的概率为 0.2。变量消除算法正确计算出 sunnyToday 为 true 的概率是 0.2。

详细说来，您刚刚看到的命令完成好几件工作：首先创建变量消除算法的一个实例，告诉实例查询目标是 sunnyToday。然后运行算法并返回 sunnyToday 值为 true 的概率。这条命令还负责执行完毕的清理，释放算法所用的任何资源。

2.2.3 构建模型和生成观测值

现在，我们开始构建一个更有趣的模型。您需要一个 Figaro 结构——If，因此要导入它。还需要一个名为 Select 的结构，但是这已经随着 com.cra.figaro.language 导入：

```
import com.cra.figaro.library.compound.If
```

我们使用 If 和 Select 构建更复杂的元素：

```
val greetingToday = If(sunnyToday,
     Select(0.6 -> "Hello, world!", 0.4 -> "Howdy, universe!"),
     Select(0.2 -> "Hello, world!", 0.8 -> "Oh no, not again"))
```

对此的思维方式是元素代表一个随机过程。在本例中，名为 greetingToday 的元素代表着这样的过程：首先检查 sunnyToday 的值，如果为 true，选择“Hello，world!”的概率

为 0.6,“Howdy, universe!”的概率为 0.4。如果 sunnyToday 的值为 false，选择“Hello, world!”的概率为 0.2，“Oh no, not again”的概率为 0.8。greetingToday 是一个复合元素，因为它由 3 个元素构建而成。由于 greetingToday 的可能取值为字符串，所以它是 Element[String]。

现在，假定您已经看到今天的问候语是“Hello, world!”，您可以使用一个观测值说明这一证据：

```
greetingToday.observe("Hello, world!")
```

接下来，您可以根据问候语是“Hello, world!”算出今天是晴天的概率：

```
println(VariableElimination.probability(sunnyToday, true))
```

这条命令打印输出 0.4285714285714285。注意，结果明显高于前一个答案（0.2）。这是因为问候语是“Hello, world!”时，今天是晴天的可能性高于其他情况，所以证据支持今天是晴天。这一推理是贝叶斯法则的简单实例，第 9 章将介绍这一法则。

您打算扩展该模型，用不同证据运行更多查询，所以应该移除变量 greetingToday 的观测值。用如下命令可以完成：

```
greetingToday.unobserve()
```

现在，如果发出如下查询：

```
println(VariableElimination.probability(sunnyToday, true))
```

您将得到和指定证据之前一样的答案 0.2。

在本节的最后，我们进一步细化模型：

```
val sunnyTomorrow = If(sunnyToday, Flip(0.8), Flip(0.05))
val greetingTomorrow = If(sunnyTomorrow,
     Select(0.6 -> "Hello, world!", 0.4 -> "Howdy, universe!"),
     Select(0.2 -> "Hello, world!", 0.8 -> "Oh no, not again"))
```

您可以计算在有无关于今天问候语的证据情况下，明天的问候语为“Hello, world!”的概率：

```
println(VariableElimination.probability(greetingTomorrow, "Hello, world!"))
// prints 0.27999999999999997

greetingToday.observe("Hello, world!")
println(VariableElimination.probability(greetingTomorrow, "Hello, world!"))
// prints 0.3485714285714286
```

可以看到，在观察到今天的问候语是“Hello, world!”时，明天的问候语是“Hello, world!”的概率增大，为什么？因为今天的问候语是“Hello, world!”，今天就更有可能是晴天，明天是晴天的可能性也就更大，最终使明天的问候语更可能是“Hello, world!”。正如在第 1 章中所看到的，这是推断过去更好预测未来的一个例子，Figaro 负责所有的计算。

2.2.4　理解模型的构建方法

现在，您已经看到了创建模型、指定证据和查询、运行推理得到答案的所有步骤，接下来我们更仔细地观察 Hello World 模型，理解如何从构件（原子元素）和连接器（复合元素）构建它。

图 2-4 是模型的图形描述。该图首先重现了模型定义，每个 Scala 变量在一个单独的方

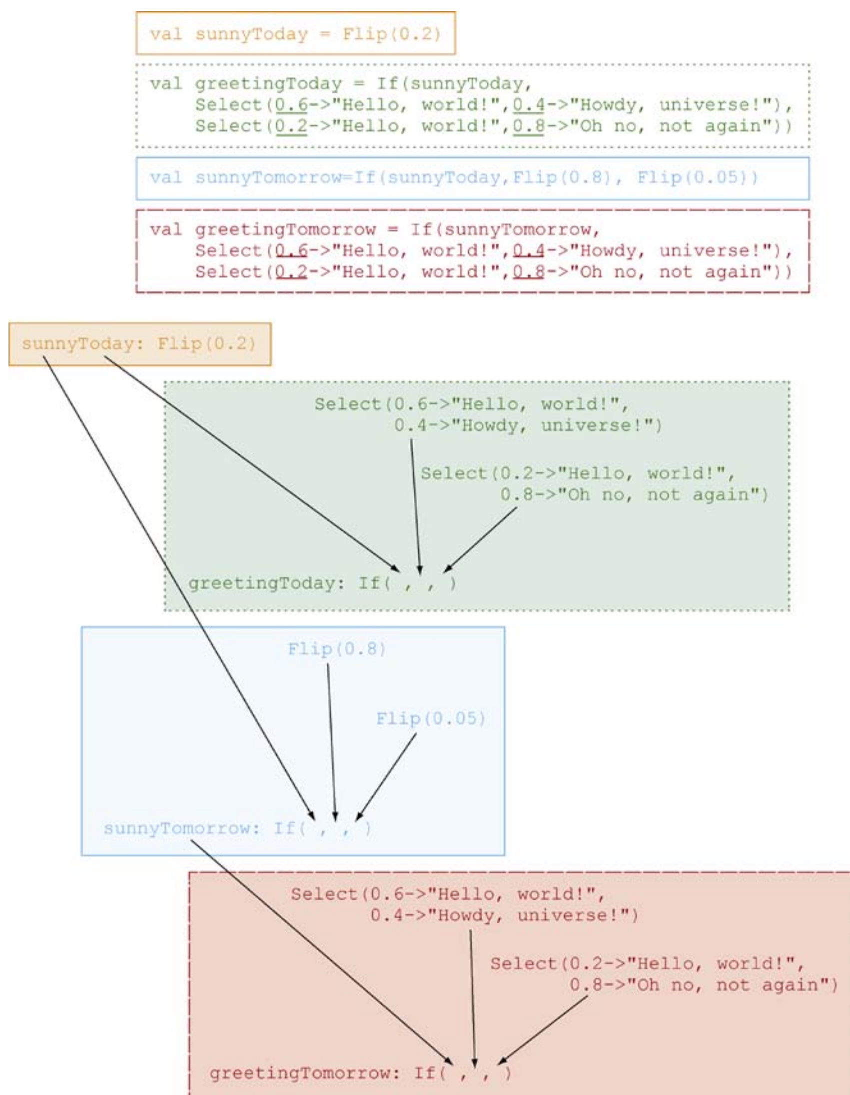


图 2-4　以图的形式显示的 Hello World 模型结构。图中的每个节点是一个元素。图中的边显示何时某个元素被另一个元素使用

框中。在下半部分中，每个节点表示模型中的对应元素，同样在单独的方框中显示。有些元素本身就是 Scala 变量值。例如，Scala 变量 sunnyToday 的值是 Flip(0.2)元素。如果元素是 Scala 变量值，图上显示变量名称和元素。该模型还包含了一些不是特定 Scala 变量值，但仍出现在模型中的元素。例如，因为 sunnyTomorrow 的定义是 If(sunnyToday, Flip(0.8), Flip(0.05))，Flip(0.8) 和 Flip(0.05)也是模型的一部分，所以它们显示为图中的节点。

该图包含了元素之间的边。例如，从 Flip(0.8)到取 sunnyTomorrow 值的 If 元素之间有一条边，表明 If 元素使用 Flip(0.8)元素。一般来说，如果第二个元素的定义中使用了第一个元素，则两者之间存在一条边。因为只有复合元素是由其他元素构建而成的，所以只有复合元素可能成为边的终点。

2.2.5 理解重复的元素：何时相同，何时不同

需要注意的一点是，Select(0.6 -> "Hello, world!", 0.4 -> "Howdy, universe!")在图中出现了两次，Select(0.2 -> "Hello, world!", 0.8 -> "Oh no, not again")也是如此。这是因为代码中定义出现了两次，一次用于 greetingToday，另一次用于 greetingTomorrow。尽管定义相同，但是这是两个不同的元素。它们在 Figaro 模型定义的随机过程的同一次执行中可能取不同的值。例如，该元素的第一个实例可能取值“Hello, world!”，而第二个实例可能取值“Howdy, universe!”。这是有意义的，因为第一个元素实例用于定义 greetingToday，第二个则用于定义 greetingTomorrow。今天和明天的问候语很可能不一样。

这和常规编程类似，想象一下您有一个 Greeting 类和如下代码：

```
class Greeting {
  var string = "Hello, world!"
}
val greetingToday = new Greeting
val greetingTomorrow = new Greeting
greetingTomorrow.string = "Howdy, universe!"
```

尽管定义完全相同，greetingToday 和 greetingTomorrow 是 Greeting 类的两个不同实例。因此，greetingTomorrow.string 和 greetingToday.string 可能取不同值，后者仍然等于“Hello, world!”。同样，Figaro 构造函数（如 Select）创建对应元素类的新实例。所以 greetingToday 和 greetingTomorrow 是 Select 元素的两个不同实例，因此在一次运行中可能取不同的值。

另一方面，注意 Scala 变量 sunnyToday 也出现了两次，一次在 greetingToday 的定义中，另一次在 sunnyTomorrow 中。但是本身是 sunnyToday 值的元素在图中仅出现一次。为什么？因为 sunnyToday 是一个 Scala 变量，而不是 Figaro 元素定义。当 Scala 变量在一段代码中出现超过一次时，它是相同的变量，所以使用相同的值。在我们的模型中，

这是有意义的；它表示同一天的天气，用于 greetingToday 和 sunnyTomorrow 的定义中，所以在模型的任何随机执行中都取相同的值。

常规代码中也会发生相同的事情。如果编写如下代码：

```
val greetingToday = new Greeting
val anotherGreetingToday = greetingToday
anotherGreetingToday.string = "Howdy, universe!"
```

anotherGreetingToday 和 greetingToday 是相同的 Scala 变量，所以运行上述代码之后，greetingToday 的值也是“Howdy, universe!”，同样，如果同一个 Scala 变量代表程序中出现多次的一个元素，它在每次运行中也取相同的值。

理解这一点对于了解 Figaro 模型的构建方式是必不可少的，所以我建议反复阅读本节以确保理解。此时，您应该已经概要了解所有的 Figaro 主要概念以及它们的组合方式。在下面几节中，您将更详细地研究其中一些概念，下一节首先介绍原子元素。

2.3 使用基本构件：原子元素

现在正是积累 Figaro 元素知识的时机。我将首先介绍模型的基本构件——原子元素。**原子**这一名称意味着，它们不依赖于任何其他元素，完全是独立的。在此我不提供完整的原子元素列表，只介绍最常见的例子。

原子元素根据值的类型分为离散元素和连续元素。**离散**原子元素取 Boolean 和 Integer 等类型的值，而**连续**原子元素通常使用 Double 类型的值。从技术上说，离散意味着值之间有很清晰的分隔。例如，整数 1 和 2 有很清晰的分隔，其中没有任何整数。而连续意味着值处于一个没有分隔的连续域中，如实数。在任何两个实数之间都有更多的实数。离散和连续元素之间的差异造成了概率定义的差异，第 4 章中将做介绍。

警告：有些人认为**离散**就意味着**有限**。这是错误的。例如，整数有无穷多个，但是它们是清晰分隔的，所以是离散值。

关键定义

原子元素——不依赖于任何其他元素的独立元素。
复合元素——由其他元素构成的元素。
离散元素——值类型清晰分隔的元素。
连续元素——值类型没有分隔的元素。

2.3.1 离散原子元素

让我们来看一些离散原子元素的例子：Flip、Select 和 Binomial。

Flip

您已经看到了离散原子元素 Flip。Flip 包含在 com.cra.figaro.language 包中，该包中有许多最常用的元素。我建议始终在程序开始处导入该包的所有内容。一般来说，Flip 取一个参数 *p*，表示元素值为真的概率。*p* 应该是 0 和 1(包含)之间的数值。该元素值为假的概率是 1-*p*。例如：

```
import com.cra.figaro.language._
val sunnyToday = Flip(0.2)
println(VariableElimination.probability(sunnyToday, true))
// prints 0.2

println(VariableElimination.probability(sunnyToday, false))
// prints 0.8
```

Flip(0.2)的正式类型是 AtomicFlip，是 Element[Boolean]的子类。这使其区别于后面将会看到的 CompoundFlip。

Select

您已经在 Hello World 程序中看到了 Select 元素。下面是一个例子：Select(0.6 -> "Hello, world!", 0.3 -> "Howdy, universe!", 0.1 -> "Oh no, not again")。图 2-5 展示了这个元素的构成。在圆括号中是一些子句。每个子句由一个概率、一个右箭头和一个可能结果组成。子句的数量是可变的，您可以使用任意个子句。图中有 3 个子句。因为所有结果的类型都是 String，所以这个元素是 Element[String]。同样，其正式类型是 AtomicSelect[String]——Element[String]的子类。

很自然，Select 元素对应于一个过程，其中每个可能的结果按照对应的概率而选择。下面是其工作方式：

```
val greeting = Select(0.6 -> "Hello, world!", 0.3 -> "Howdy, universe!", 0.1
    -> "Oh no, not again")
println(VariableElimination.probability(greeting, "Howdy, universe!"))
// prints 0.30000000000000004
```

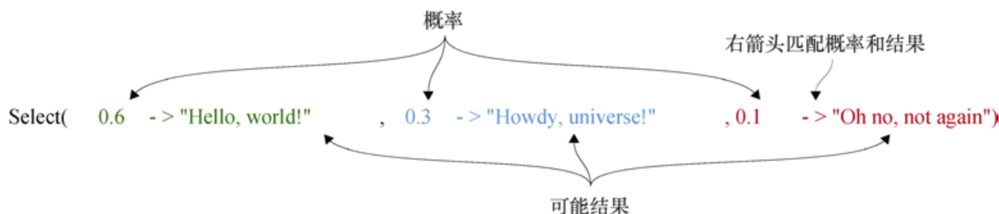


图 2-5 Select 元素结构

注意，在 Select 中，概率累加起来不一定等于 1。如果它们加起来不等于 1，概率将被**规格化**——加起来等于 1，同时保持概率之间的比例。在下面的例子中，每个概率都等于前一个例子中的两倍，因此加总起来等于 2。规格化之后恢复成前一个例子中的概率，因此得到相同的结果。

```
val greeting = Select(1.2 -> "Hello, world!", 0.6 -> "Howdy, universe!", 0.2
     -> "Oh no, not again")
println(VariableElimination.probability(greeting, "Howdy, universe!"))
//prints 0.30000000000000004
```

Binomial

Binomial 是一个有用的离散元素。想象一下一周有 7 天，每天都有一个“晴天”元素 Flip（0.2）。现在您想要一个元素，其值为一周中放晴的天数。这可以用元素 Binomial(7, 0.2)实现。这个元素的值是总共尝试 7 次，每次尝试为 true 的概率为 0.2 的情况下，尝试结果为 true 的次数。可以这样使用它：

```
import com.cra.figaro.library.atomic.discrete.Binomial
val numSunnyDaysInWeek = Binomial(7, 0.2)
   println(VariableElimination.probability(numSunnyDaysInWeek, 3))
//prints 0.114688
```

一般来说，Binomial 取两个参数：尝试次数和每次尝试得出结果 true 的概率。Binomial 的定义假定所有尝试是独立的，第一次尝试为真不会改变第二次尝试得出 true 的概率。

2.3.2　连续原子元素

本节介绍两个连续原子元素的常见例子——Normal 和 Uniform。第 4 章详细说明连续元素。连续概率分布与离散分布略有不同，指定的不是每个值的概率，而是每个值的**概率密度**，概率密度描述的是以该值为中心的每个区间的概率。您仍然可以将概率密度视为和常规概率类似的概念，表明某个值与其他值可能性的对比。因为本章是 Figaro 的教程，第 4 章解释概率模型，我将把进一步的讨论推迟到那个时候。请放心，在后面这一点将会更加清晰。

Normal

正态分布是您可能熟悉的一种连续概率分布。**正态分布**还有其他一些名称，包括**钟形曲线**和**高斯分布**。图 2-6 展示了正态分布的概率密度函数。（如果非要吹毛求疵，可能称之为“**单变量正态分布**”更为合适，因为它定义了单一实数变量上的概率，您还可以在多个变量上定义多变量正态分布，但是我们不是那样的人，所以就称之为正态分布。）这个函数有一个**均值**——中心点（图中为 1），以及一个**标准差**——函数沿中心点

分布的程度（图中为 0.5）。大约有 68%的情况下，从正态分布生成一个值将得到与均值相差一个标准差的区间内的值。在统计和概率推理中，正态分布通常用均值和**方差**描述，后者是标准差的平方。图中的标准差为 0.5，方差为 0.25。因此，这个特定正态分布的标准规格描述是 Normal（1,0.25）。

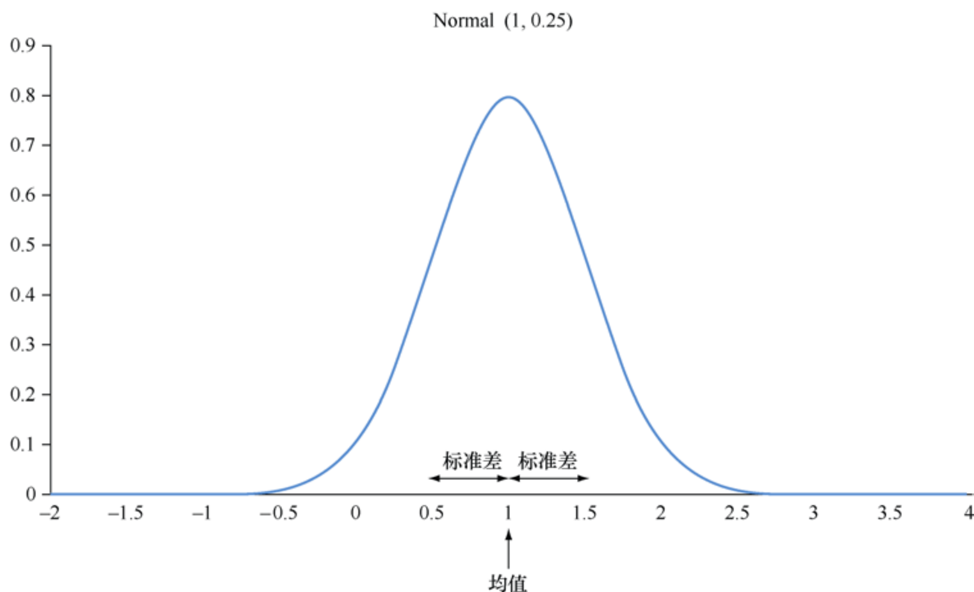


图 2-6 正态分布的概率密度函数

Figaro 遵循上述约定。它提供一个 Normal 元素，以均值和方差作为参数。可以这样定义 Normal 元素：

```
import com.cra.figaro.library.atomic.continuous.Normal
val temperature = Normal(40, 100)
```

均值为 40，方差为 100，意味着标准差为 10。现在，假定您想要用这一元素进行推理。Figaro 的变量消除算法只适用于可能取值个数有限的元素。特别是，它不能用于连续元素。所以，需要一个不同的算法。您将使用称作**重要性抽样**的算法，这是一种很适合于连续元素的近似算法。算法的运行方法如下：

```
import com.cra.figaro.algorithm.sampling.Importance
def greaterThan50(d: Double) = d > 50
println(Importance.probability(temperature, greaterThan50 _))
```

重要性抽样是一种每次产生不同答案的随机算法，这些答案通常应该在真值的附近。您得到的答案应该与 0.1567 接近，但是很有可能得到稍有不同的答案。

注意，这里的查询和之前略有不同。对于连续元素，取特定值（如 50）的概率通常为 0。原因是，连续元素中有无穷多个没有分隔的值。该过程得出 50 而非 50.000000000000001 或者之间其他值的可能性无限小。所以，通常不要向连续元素发出特定值概率的查询。

相反，您可以查询该值落入某个区间的概率。例子中的查询是预测 greaterThan50，这个预测以双精度（Double）类型值作为参数，如果参数大于 50 则返回 true。预测是一个元素值的布尔函数。当您查询某个元素是否满足预测时，询问的是将预测应用到某个元素值返回 true 的概率。在这个例子中，您的查询计算温度高于 50 的概率。

Scala 注释：greaterThan50 之后的下划线告诉 Scala，greaterThan50 是一个传递给 Importance.probability 方法的函数值。如果没有这个下划线，Scala 可能认为您试图将该函数应用到 0 参数上，从而出错。有时候，Scala 可能自动理解这一点而无需明确地提供下划线。但有时候它无法做到，而会告诉您提供下划线。

Uniform

我们再来介绍一个同样熟悉的连续元素例子。**Uniform 元素**取指定区间中的值，区间中的每个值可能性相同。您可以这样创建和使用 Uniform 元素：

```
import com.cra.figaro.library.atomic.continuous.Uniform
val temperature = Uniform(10, 70)
Importance.probability(temperature, greaterThan50 _)
// prints something like 0.3334
```

Uniform 元素取两个参数：最小值和最大值。从最小值到最大值的所有值概率密度相同。在前一个例子中，最小值为 10，最大值为 70，所以范围的大小为 60。您的查询预测是该值是否在 50～70——区间大小为 20。所以，预测的概率为 20/60 或者 1/3，可以看到，重要性抽样得到的结果与此接近。

最后说明一下：这个元素的官方名称是**连续均匀分布**（continuous uniform）。在 com.cra.figaro.library.atomic.discrete 包中还可以找到离散均匀分布。正如您的预期，离散均匀分布明确列出一组值，其中的每个值出现可能性相同。

好了，现在您已经了解了构件，下面我们来看看如何组合它们，创建更大的模型。

2.4　使用复合元素组合原子元素

在本节中，您将看到一些复合元素。前面已经说过，复合元素是构建于其他元素之上的更为复杂的元素。复合元素的例子很多，您将首先考察两个特殊的例子，If 和 Dist，然后了解如何使用大部分原子元素的复合版本。

2.4.1 If

您已经看到了 If 复合元素的一个例子。它由 3 个元素组成：一个测试，一个 then 子句和一个 else 子句。If 表示这样的随机过程：首先检查测试的结果，如果测试值为 true，则生成 then 子句的值；否则，生成 else 子句的值。图 2-7 展示了 If 元素的一个例子。If 取 3 个参数，第一个参数是 Element[Boolean]，例中是 sunnyToday 元素。这个参数表示测试。第二个参数是 then 子句，当测试元素的值为 true，则选中这个元素。如果测试元素值为 false，则选择第三个参数——else 子句。then 和 else 子句必须有相同的值类型，这将成为 If 元素的值类型。

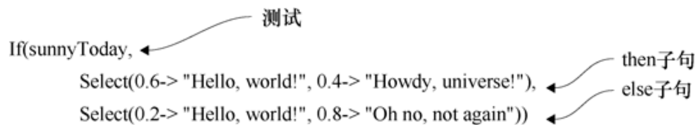


图 2-7 If 元素结构

下面是 If 元素的执行情况：

```
val sunnyToday = Flip(0.2)
val greetingToday = If(sunnyToday,
     Select(0.6 -> "Hello, world!", 0.4 -> "Howdy, universe!"),
     Select(0.2 -> "Hello, world!", 0.8 -> "Oh no, not again"))
println(VariableElimination.probability(greetingToday, "Hello world!"))
// prints 0.27999999999999997
```

上述代码打印输出 0.28（在舍入误差范围内）是因为 then 子句被选中的概率为 0.2（当 sunnyToday 为 true），而这种情况下“Hello, world!”出现的概率为 0.6。同时，else 子句被选中的概率为 0.8，此时“Hello, world!”出现的概率为 0.2。所以，“Hello, world!”出现的总概率为(0.2 × 0.6) + (0.8 × 0.2) = 0.28。您可以通过明确评估两种情况看到这一结果：

```
sunnyToday.observe(true)
println(VariableElimination.probability(greetingToday, "Hello, world!"))
// prints 0.6 because the then clause is always taken

sunnyToday.observe(false)
println(VariableElimination.probability(greetingToday, "Hello, world!"))
// prints 0.2 because the else clause is always taken
```

2.4.2 Dist

Dist 是一个实用的复合元素，Dist 与 Select 类似，但是它选择一组元素中的一个而不是选择一组值中的一个。每个选择本身是一个元素，这也就是 Dist 是复合元素的原因。如果您想要在本身是随机过程的复杂选择之间选择，Dist 很实用。例如，足球中的角球可能从短传或者高吊传中开始。这可以由一个 Dist 元素表示，该元素在两个过程之间选择，一个从短传开始，另一个从高吊传中开始。

Dist 在 com.cra.figaro.language 包中，下面是 Dist 元素的一个例子：

```
val goodMood = Dist(0.2 -> Flip(0.6), 0.8 -> Flip(0.2))
```

如图 2-8 所示，Dist 元素的结构类似于图 2-5 中 Select 的结构。不同之处在于，选择的不是结果值而是结果元素。您可以这样认为：Select 直接选择一组可能值中的一个，没有任何中间过程。Dist 是间接的：它选择一组过程中的一个运行，在运行中产生一个值。在例子中，由 Flip(0.6)表示的过程被选中的概率为 0.2，由 Flip(0.2)表示的过程被选中的概率为 0.8。

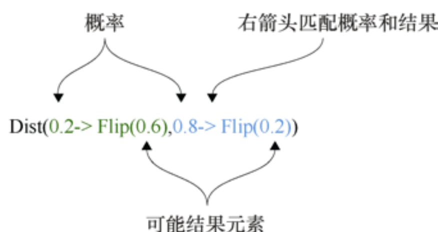


图 2-8 Dist 元素结构

下面是查询这一元素得到的结果：

```
println (VariableElimination.probability(goodMood, true))
// prints 0.28 = 0.2 * 0.6 + 0.8 * 0.2
```

2.4.3 原子元素的复合版本

您已经看到了采用数值参数的原子元素。例如，Flip 以概率为参数，Normal 以均值和方差为参数。如果您对这些数值参数不确定该怎么办？在 Figaro 中，答案很简单：将它们自身作为元素。

制作数值参数元素时，您就得到了原子元素的复合版本。例如，下面的代码定义一个复合 Flip 元素，并用它进行推理：

```
val sunnyTodayProbability = Uniform(0, 0.5)
val sunnyToday = Flip(sunnyTodayProbability)
println(Importance.probability(sunnyToday, true))
// prints something like 0.2548
```

这里，sunnyTodayProbability 表示对今天是晴天的概率的不确定性，您认为 0～0.5 的任何值可能性一样大。那么，sunnyToday 为 true 的概率等于 sunnyTodayProbability 元素的值。一般来说，复合 Flip 取单一参数，即表示 Flip 为 true 概率的一个 Element[Double]。

Normal 提供了更多可能性。在概率推理中，假定正态分布有一个特定的已知方差，通过均值表示不确定性的情况很常见。例如，您可能假定温度的方差为 100，而均值在 40 左右，但是对此不太确定。此时可以用如下代码捕捉这种不确定性：

```
val tempMean = Normal(40, 9)
val temperature = Normal(tempMean, 100)
println(Importance.probability(temperature, (d: Double) => d > 50))
// prints something like 0.164
```

此外，您也可能对方差不确定，认为它可能是 80 或者 105。此时可以使用如下代码：

```
val tempMean = Normal(40, 9)
val tempVariance = Select(0.5 -> 80.0, 0.5 -> 105.0)
val temperature = Normal(tempMean, tempVariance)
println(Importance.probability(temperature, (d: Double) => d > 50)
// prints something like 0.1549
```

更多信息请参见：本章只说明了 Figaro 提供的原子和复合元素中的一小部分。您可以查看 Figaro 的 Scaladoc，以了解更多实用的例子。Scaladoc（www.cra.com/Figaro）和 Javadoc 类似，是自动生成的 Figaro 库 HTML 文档。它还包含在 Figaro 二进制下载中，可从 Figaro 网页上下载。这一下载包含一个形如 figaro_2.11-2.2.2.0-javadoc.jar 的文件。该档案的内容可以使用 7-Zip 或者 WinZip 等程序解压。

2.5 用 Apply 和 Chain 构建更复杂的模型

Figaro 提供两种构建模型的有用工具，称作 Apply 和 Chain。这两种工具都是重要的元素。Apply 可以在 Figaro 中引入 Scala，利用其全部能力。Chain 可以无限的方式创建元素之间的相互依赖关系。目前为止您所看到的复合元素（如 If 和复合 Flip）可以特定的预定义方式创建依赖关系。Chain 可以超越这些预定义依赖关系，创建您所需要的任何依赖。

2.5.1　Apply

我们从 Apply 开始，这个元素在 com.cra.figaro.language 包中。Apply 以一个元素和一个 Scala 函数作为参数，它代表将 Scala 函数应用到该元素值以获得新值的过程。例如：

```
val sunnyDaysInMonth = Binomial(30, 0.2)
def getQuality(i: Int): String =
  if (i > 10) "good"; else if (i > 5) "average"; else "poor"
val monthQuality = Apply(sunnyDaysInMonth, getQuality)
println(VariableElimination.probability(monthQuality, "good"))
// prints 0.025616255335326698
```

上述代码中的第 2 行和第 3 行定义一个名为 getQuality 的函数。这个函数取一个 Integer 型参数，getQuality 函数中的参数局部名称为 i。根据第 3 行中的代码，该函数返回一个字符串。

第 4 行定义一个 Apply 元素 monthQuality。Apply 元素的结构如图 2-9 所示，它取两个参数，第一个是元素，在本例中是 Element[Int]类型的 sunnyDaysInMonth。第二个参数是函数，参数类型与元素的值类型相同。在我们的例子中，函数 getQuality 取得一个 Integer 型参数，因此两者互相匹配。该函数可以返回任何类型的值，在我们的例子中，函数返回一个字符串。



图 2-9　Apply 元素结构

下面介绍 Apply 元素定义随机过程的方式。它首先生成第一个元素参数的值。在我们的例子中，生成当月中晴天的特定数量，我们假定生成的是 7。然后，该过程取得第二个函数参数，并将其应用到生成的值。在我们的例子中，过程将函数 getQuality 应用到 7，获得结果 average。该结果成为 Apply 元素的值。从这里可以看出，Apply 的可能值和函数参数的返回值类型相同。在我们的例子中，Apply 元素是一个 Element[String]。

如果您对 Scala 还很陌生，那么我来介绍一下**匿名函数**。如果只想在一个位置上使用，为每个 Apply 元素定义单独的函数可能令人烦恼，尤其是上例中的简短函数。Scala 提供了匿名函数，可以直接在使用的位置定义。图 2-10 展示了一个匿名函数的结构，它的定义和 getQuality 相同。这一结构的组件类似于命名函数。函数有一个参数 i，类型为 Integer。=>符号表示定义的是匿名函数。最后是一个函数体，与 getQuality 相同。

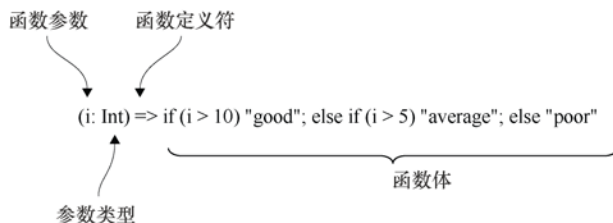


图 2-10 匿名函数结构

您可以使用匿名函数定义和前一个元素等价的 Apply 元素：

```
val monthQuality = Apply(sunnyDaysInMonth,
  (i: Int) => if (i > 10) "good"; else if (i > 5) "average"; else "poor")
```

现在，您可以查询 monthQuality。不管使用哪一个版本的 monthQuality，得到的答案都相同：

```
println(VariableElimination.probability(monthQuality, "good"))
// prints 0.025616255335326698
```

虽然 Apply 的这个例子是人为的，但是使用它有许多实际的理由。下面是几个例子。

- 您有一个双精度元素，希望将其值舍入为最近的整数。
- 您有一个元素的值类型是一个数据结构，希望概括该数据结构的一个属性。您可能有一个以列表为基础的元素，希望知道列表项目数量大于 10 的概率。
- 两个元素之间的关系最好由一个物理学模型编码。在这种情况下，您可以使用一个 Scala 函数表示物理关系，使用 Apply 将物理模型嵌入 Figaro。

多参数 Apply

使用 Apply 时，并不限于只有一个参数的 Scala 函数。Figaro 中定义的 Apply 可以使用最多 5 个参数。使用多于一个参数的 Apply，是将多个元素结合在一起承载另一个元素的好办法。例如，下面是两个参数的 Apply：

```
val teamWinsInMonth = Binomial(5, 0.4)
val monthQuality = Apply(sunnyDaysInMonth, teamWinsInMonth,
     (days: Int, wins: Int) => {
     val x = days * wins
     if (x > 20) "good"; else if (x > 10) "average"; else "poor"
     })
```

这里，Apply 的两个元素参数是 sunnyDaysInMonth 和 teamWinsInMonth，它们都是 Element[Int]。函数参数有名为 days 和 wins 的 Integer 参数。这个函数创建一个局部变量 x，其值等于 days * wins。注意，因为 days 和 wins 是常规的 Scala Integer 变量，x 也是常规 Scala 变量，而不是 Figaro 元素。实际上，Apply 函数参数中的任何东西都是常规的 Scala 内容。Apply 取得在常规 Scala 值上操作的 Scala 函数，将其“提升”为在 Figaro 元素上操作的函数。

现在，查询这个版本的 monthQuality：

```
println(VariableElimination.probability(monthQuality, "good"))
// prints 0.15100056576418375
```

得出的概率值略微提升。似乎，我的垒球队有机会振奋人心。更重要的是，尽管这是一个简单的例子，但是例中的概率在没有 Figaro 的情况下也很难计算。

2.5.2　Chain

顾名思义，Chain（链）用于将元素链接为一个模型，模型中的元素依赖于另一个元素，那个元素又依赖其他的元素，依次类推。这与概率的链式法则相关，第 5 章将介绍这一法则。但是，理解 Chain 并不需要知道链式法则。

Chain 也包含在 com.cra.figaro.language 包中。解释 Chain 的最简单方式是通过一张图。图 2-11 展示了两个元素：goodMood 元素依赖 monthQuality 元素。如果您将其看作一个随机过程，该过程首先生成 monthQuality 的值，然后使用该值生成 goodMood 的值。这是贝叶斯网络的一个简单例子，第 4 章中您将学习这种方法。图中借用了贝叶斯网络的术语：monthQuality 称为**父节点**，goodMood 称为**子节点**。

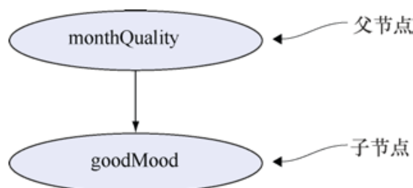


图 2-11　一个变量依赖于另一个变量的双变量模型

因为 goodMood 依赖于 monthQuality，goodMood 使用 Chain 定义。monthQuality 元素在前一小节已经定义。下面是 goodMood 的定义：

```
val goodMood = Chain(monthQuality, (s: String) =>
     if (s == "good") Flip(0.9)
     else if (s == "average") Flip(0.6)
     else Flip(0.1))
```

图 2-12 展示了这个元素的结构。和 Apply 类似，Chain 取两个参数：一个元素和一个函数。在本例中，元素是父节点，函数被称为**链函数**。Chain 和 Apply 之间的差别是 Apply 中的函数返回常规的 Scala 值，而 Chain 中的函数返回一个元素。在本例中，函数返回一个 Flip，选择哪一个 Flip 取决于 monthQuality 的值。所以，这个函数取得类型为字符串的参数，返回一个 Element[Boolean]。

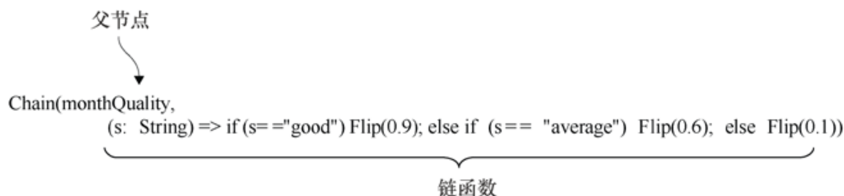


图 2-12　Chain 元素结构

这个 Chain 元素定义的随机过程如图 2-13 所示。这一过程有 3 个步骤。首先，为父节点生成一个值。在本例中，为 monthQuality 生成值 average。其次，对该值应用链函数以获得一个元素，该元素称作**结果元素**。在例子中，您可以检查链函数的定义，发现结果元素是 Flip(0.6)。第三，从结果元素生成一个值，例子中生成的是 true。这个值成为子节点的值。

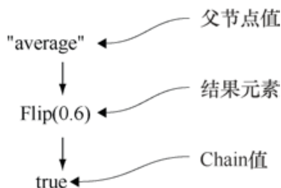


图 2-13　Chain 元素定义的随机过程。首先，为父节点生成一个值。接下来，根据链函数选择结果元素。最后，从结果元素生成一个值

下面我们总结 Chain 中涉及的所有类型。Chain 由两个类型参数化——父节点的值类型（称作 T）和子节点的值类型（称作 U）。

- **父节点**类型为 Element[T]。
- **父节点值**类型为 T。
- **链函数**类型为 T => Element[U]。这意味着该函数从 T 类型得出 Element[U]。
- **结果元素**的类型为 Element[U]。
- **链值**类型为 U。
- **子节点**的类型为 Element[U]。这是整个 Chain 元素的类型。

在我们的例子中，goodMood 是 Element[Boolean]，可以查询其值为 true 的概率：

```
println(VariableElimination.probability(goodMood, true))
// prints 0.3939286578054374
```

多参数 Chain

让我们来考虑一个稍微复杂些的模型，其中 goodMood 依赖 monthQuality 和 sunnyToday，如图 2-14 所示。

可以使用一个双参数的 Chain 捕捉上述事实。在本例中，链中的函数有两个参数——名为 quality 的字符串参数和名为 sunny 的布尔型参数，返回一个 Element[Boolean]。goodMood 同样是 Element[Boolean]。

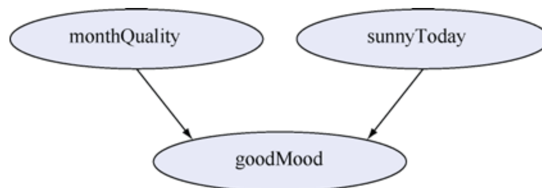


图 2-14　一个三变量模型，其中 goodMood 取决于其他两个变量

下面是代码：

```
val sunnyToday = Flip(0.2)
val goodMood = Chain(monthQuality, sunnyToday,
  (quality: String, sunny: Boolean) =>
    if (sunny) {
      if (quality == "good") Flip(0.9)
      else if (quality == "average") Flip(0.7)
      else Flip(0.4)
    } else {
      if (quality == "good") Flip(0.6)
      else if (quality == "average") Flip(0.3)
      else Flip(0.05)
    })
println(VariableElimination.probability(goodMood, true))
// prints 0.2896316752495942
```

注意：和 Apply 不同，Chain 结构仅定义为一个或者两个参数。如果需要更多参数，可以结合 Chain 和 Apply。首先使用 Apply 将参数元素打包为单一元素，该元素的取值是参数值的元组。将这个元素传递给 Chain。这样 Chain 就得到了元素求值中需要的所有信息。

使用 Apply 和 Chain 的 map 及 flatMap

熟悉 Scala 的读者请注意，在某种程度上，Figaro 元素类似于 Scala 集合（如列表）。列表包含一组值，而元素包含一个随机值。正如可以对列表中的每个值应用函数以获得新列表那样，您可以对包含在元素中的随机值应用函数以得到一个新元素。这正是 Apply 所做的！对于列表，对列表中的每个值应用某个函数是通过使用 map 方法实现的。类似地，使用 Apply 就为元素定义了 map 操作。因此，可以将 Apply(Flip(0.2), (b: Boolean) => !b)写作 Flip(0.2).map(!_)。

同样，对于列表，可以对每个值应用某个函数返回列表，然后用 flatMap 将所有结果列表扁平化为单一列表。Chain 以同样的方式对元素中包含的随机值应用函数，获得另一个元素，然后取出元素中的值。所以，元素的 flatMap 用 Chain 定义。因此，您可以将 Chain(Uniform(0, 0.5), (d: Double) => Flip(d))写做 Uniform(0, 0.5).flatMap(Flip(_))。（顺便说一句，要注意使用 Chain 定义复合 Flip 的方式。许多 Figaro 复合元素可以用 Chain 定义。）

Scala 中最棒的特性之一是任何定义了 map 和 flatMap 的类型都可以用于 for 循环。可以对元素使用 for 标记。

可以编写如下代码：

```
for { winProb <- Uniform(0, 0.5); win <- Flip(winProb) } yield !win
```

这就是我们用简单元素构建复杂模型的一段旅程。在结束本章之前，我将描述一些应用证据和约束模型的方法。

2.6 使用条件和约束指定证据

现在，我已经相当详细地介绍了构建模型的方法。使用 Figaro 时，您的很大一部分精力将花在创建模型上。但是不要忽略证据的指定。Figaro 提供了 3 种说明证据的机制：观测值，条件和约束。

2.6.1 观测值

您已经看到使用观测值指定证据的方法。在 Hello World 程序中已经提供了一个例子，使用如下的语句：

```
greetingToday.observe("Hello, world!")
```

一般来说，observe 是元素上定义的一个方法，以元素的某个可能取值作为参数。它的效果是规定该元素必须采用该值。元素采用不同值的任何随机执行都被排除。这个

观测值对相关元素的概率有影响。例如，在 Hello World 程序中，您可以看到声明这个观测值使 sunnyToday 更可能为 true，因为根据该模型，今天是晴天比不是晴天更可能造成“Hello, world!”的问候语。

您还看到了关联观测值的另一种方法：

```
greetingToday.unobserve()
```

上述语句删除 greetingToday 上的观测值（如果有的话），这样就不会排除任何随机执行。结果是，关于 greetingToday 的证据不影响其他元素的概率。

2.6.2 条件

observe 指定元素的特定值。如果您知道元素值的某些相关情况，但是不确定该值，该怎么办？Figaro 允许将任何预测作为证据。这称作**条件**。条件指定一个布尔函数，该函数为真某个值才可能出现。例如：

```
val sunnyDaysInMonth = Binomial(30, 0.2)
println(VariableElimination.probability(sunnyDaysInMonth, 5))
sunnyDaysInMonth.setCondition((i: Int) => i > 8)
println(VariableElimination.probability(sunnyDaysInMonth, 5))
```

打印 0.172279182850003

晴天超过 8 天的证据

打印 0，因为 5 个晴天与证据不符

观察证据并用它推断其他变量的概率是概率推理的核心。例如，通过观察当月的晴天数量，可以推断某人是否可能拥有好心情。使用来自 2.5 节的例子：

```
val sunnyDaysInMonth = Binomial(30, 0.2)
val monthQuality = Apply(sunnyDaysInMonth,
  (i: Int) => if (i > 10) "good"; else if (i > 5) "average"; else "poor")
val goodMood = Chain(monthQuality, (s: String) =>
     if (s == "good") Flip(0.9)
     else if (s == "average") Flip(0.6)
     else Flip(0.1))
println(VariableElimination.probability(goodMood, true))
// prints 0.3939286578054374 with no evidence
```

设定条件，规定 sunnyDaysInMonth 的值必须大于 8。然后，可以看到它对 goodMood 的影响：

```
sunnyDaysInMonth.setCondition((i: Int) => i > 8)
println(VariableElimination.probability(goodMood, true))
// prints 0.6597344078195809
```

好心情的概率明显上升，因为您排除了晴天数量少于 8 的“坏”月份。

Figaro 允许指定某个元素满足的多个条件。这通过使用 addCondition 方法实现，该

方法向元素的现有条件中增添一个条件。增添条件的结果是元素值必须同时满足新条件和现有条件。

```
sunnyDaysInMonth.addCondition((i: Int) => i % 3 == 2)
```

上述语句说明，除了大于 8 之外，sunnyDaysInMonth 的值还必须比 3 的倍数大 2。这就排除了 9 和 10 等可能值，所以，最小的可能值为 11。当然，当您查询好心情的概率时，可以看到它再次上升：

```
println(VariableElimination.probability(goodMood, true))
// prints 0.9
```

可以用 removeConditions 删除某个元素上的所有条件：

```
sunnyDaysInMonth.removeConditions()
println(VariableElimination.probability(goodMood, true))
// prints 0.3939286578054374 again
```

顺便说一句，观测值只是条件的一个特例，要求某个元素取单一特定值。下面总结与条件相关的方法。

- setCondition 是元素上的一个方法，以一个预测作为参数。预测必须是从元素值类型到布尔类型的函数。setCondition 方法使这个预测成为元素上的唯一条件，删除现有条件和观测值。
- addCondition 也是元素上的方法，以预测作为参数，预测必须是从元素值类型到布尔类型的函数。addCondition 在现有条件和观测值之上添加这个预测。
- removeConditions 是元素上的方法，删除元素的所有条件和观测值。

2.6.3　约束

约束提供了规定元素相关情况的更通用手段，它通常有两个目的：（1）作为指定“软”证据的手段；（2）作为提供模型中元素间附加关系的手段。

用作软证据的约束

假定您有关于元素的某种证据，但是不是很确定。例如，您认为我看上去脾气暴躁，但是从外表无法得知我的情绪。所以您不指定 goodMood 为 false 这样的硬证据，而是指定软证据：goodMood 为 false 的可能性大于 true 的可能性。

这可以使用约束实现。**约束**是一个从元素值到 Double 类型值的函数。虽然 Figaro 没有强制规定，但是约束在这个函数值始终在 0 和 1（含）之间时工作得最好。例如，为了表示我的情绪似乎暴躁但是不确定的证据，您可以为 goodMood 添加一个约束，当 goodMood 为 true 时生成值 0.5，在 goodMood 为 false 时生成 1.0：

```
goodMood.addConstraint((b: Boolean) => if (b) 0.5; else 1.0)
```

这个软约束可以这样解读：在其他条件不变的情况下，goodMood 为 true 的可能性只有 false 的一半，因为 0.5 是 1.0 的一半。第 4 章将讨论这方面的数学运算，简而言之，某个元素值的概率乘以该值的约束结果。所以，在这个例子中，true 的概率乘以 0.5，false 的概率乘以 1.0。这样做之后，概率加总不一定等于 1，所以将被按比例调整，使之加起来等于 1。如果没有这个证据，我认为 goodMood 为 true 和 false 的可能性相同，所以它们的概率均为 0.5。看到这个证据之后，首先将两个概率乘以约束结果，得出 true 的概率为 0.25，false 的概率为 0.5，这两个数加起来不为 1，经过比例调整，最后的答案是 goodMood 为 true 的概率是 1/3，为 false 的概率是 2/3。

条件和约束之间的不同是，在条件中，声明某些状态是不可能的（概率为 0）。相比之下，约束改变不同状态的概率，但是除非结果为 0，否则不会使某个状态变为不可能。条件有时候称作**硬条件**，因为它们设置了可能状态不能违反的规则，而约束被称作**软约束**。

回到示例程序，在添加这个约束之后查询 goodMood，将看到概率因为这个软证据而下降，但是不像设置“我很暴躁”的硬证据那样为 0：

```
println(VariableElimination.probability(goodMood, true))
// prints 0.24527469450215497
```

约束提供了和条件类似的一组方法。

- setConstraint 是元素上的一个方法，以预测作为参数。预测必须是从元素值类型到 Double 类型值的函数。setConstraint 方法使该预测成为元素上的唯一约束。
- addConstraint 也是元素上的一个方法，以预测作为参数，该参数必须是从元素值类型到 Double 类型值的函数。addConstraint 方法在任何现有约束上添加这个预测。
- removeConstraints 是元素上的一个方法，从元素中删除所有约束。

条件和约束是相互独立的，所以设置条件或者删除所有条件不会删除任何现有约束，反之亦然。

作为连接元素的约束

约束有一种强大的用途。假定您认为两个元素的值相关，但是无法在元素定义中捕捉。例如，假设您的垒球队胜率为 40%，每场比赛可以通过 Flip(0.4)定义。再假设您认为自己的球队有连续性，所以相邻的比赛可能有相同的结果。您可以添加对相邻比赛的约束以捕捉这一信念，这个约束说明相邻比赛得到相同值的可能性大于得到不同值的可能性。用如下的代码可以实现上述模型。我介绍的是 3 场比赛的代码。但是可以用数组和 for 循环将其推广到任何数量的比赛。

首先，定义 3 场比赛的结果：

```
val result1 = Flip(0.4)
val result2 = Flip(0.4)
val result3 = Flip(0.4)
```

现在，为了表示发生的情况，创建一个 allWins 元素，当所有结果都为真时，其值为 true：

```
val allWins = Apply(result1, result2, result3,
  (w1: Boolean, w2: Boolean, w3: Boolean) => w1 && w2 && w3)
```

我们来看看添加任何约束之前所有比赛全胜的概率：

```
println(VariableElimination.probability(allWins, true))
// prints 0.06400000000000002
```

现在添加约束。您将定义一个 makeStreaky 函数，取得两个比赛结果并对其添加连续性约束：

```
def makeStreaky(r1: Element[Boolean], r2: Element[Boolean]) {
  val pair = Apply(r1, r2, (b1: Boolean, b2: Boolean) => (b1, b2))
     pair.setConstraint((bb: (Boolean, Boolean)) =>
    if (bb._1 == bb._2) 1.0; else 0.5
  )}
```

这个函数以两个 Boolean 元素为参数，这两个参数表示两场比赛的结果。因为约束只能应用到一个元素，您希望使用约束创建两个元素之间的关系，所以首先将两个元素打包成单一元素，该元素的值是一个二元组。这通过 Apply(r1, r2, (b1: Boolean, b2: Boolean) => (b1, b2))实现。现在，您有了一个值为两场比赛结果配对的元素。然后，设置该配对的约束为一个函数，该函数取一对 Boolean 变量 bb，如果 bb._1 ==bb._2（配对的第一部分和第二部分相等）则返回 1，否则返回 0.5。这个约束说明，在其他条件不变时，两场比赛结果相同的概率两倍于不同的概率。

现在，您可以使相邻两场比赛的结果保持延续性，并查询所有比赛全胜的概率：

```
makeStreaky(result1, result2)
makeStreaky(result2, result3)
println(VariableElimination.probability(allWins, true))
// prints 0.11034482758620691
```

可以看到，概率因为球队的连续性而明显上升。

注意：了解概率图模型的人将会注意到，使用本节中阐述的约束使 Figaro 能够描述无方向模型，如马尔科夫网络。

我们的 Figaro“旋风之旅”就要结束。在下一章中，您将看到在应用程序中使用 Figaro 的完整示例。

2.7 小结

- Figaro 使用和其他概率推理系统相同的总体结构，具有模型、证据、查询和提

供答案的推理算法。

- Figaro 模型由一组元素组成。
- Figaro 元素是表示随机过程的 Scala 数据结构。该过程生成一个值，其类型称为元素的值类型。
- Figaro 模型始于原子元素，用复合元素将原子元素组合起来。
- 可以使用 Apply 将任何 Scala 函数提升为 Figaro 模型。
- 可以使用 Chain 创建元素之间的许多有趣而复杂的依赖关系。
- 条件和约束提供指定证据和元素间附加关系的丰富框架。

2.8 练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．扩展 Hello World 程序，添加表示下床的一侧（正确或者错误）的变量。如果从错误的一侧下床，问候语总为“Oh no, not again!”，如果从正确的一侧下床，问候语的逻辑和之前相同。

2．在原始 Hello World 程序中，观测到今天的问候语是“Oh no, not again!”，查询今天的天气。现在，观测相同的证据并在练习 1 中修改的程序上提出相同的查询。查询答案发生了什么变化？能否直观地解释该结果？

3．在 Figaro 中，可以使用代码 x === z 作为如下代码的简写：

```
Apply(x, z, (b1: Boolean, b2: Boolean) => b1 === b2)
```

换言之，如果两个参数的值相等，产生一个值为 true 的元素。如果不运行 Figaro，猜测下面两段程序生成的结果：

```
a） val x = Flip(0.4)
    val y = Flip(0.4)
    val z = x
    val w = x === z
    println(VariableElimination.probability(w, true))
b） val x = Flip(0.4)
    val y = Flip(0.4)
    val z = y
    val w = x === z
    println(VariableElimination.probability(w, true))
```

现在，运行 Figaro 程序检查您的答案。

4．在下面的练习中，您将发现 FromRange 元素很有用。FromRange 有两个整数参数 *m* 和 *n*，生成 *m*～*n*-1 的随机整数。例如，FromRange(0, 3)生成 0、1、2 的概率相同。编写一段 Figaro 程序计算掷两个骰子得出总数 11 的概率。

5．编写一个 Figaro 程序计算第一个骰子掷出 6 时，两个骰子总数大于 8 的概率。

6．在“地产大亨”游戏中，当两个骰子掷出相同数字时可以多玩一个回合。如果连续三次出现这种情况，您就会入狱。编写一段 Figaro 程序计算任何一个回合中发生这种情况的概率。

7．想象一个游戏，您有一个轮盘和 5 个面数不同的骰子。轮盘有 5 个概率相等的结果：4、6、8、12 和 20。在游戏中，首先转动轮盘，然后滚动面数与轮盘结果相同的骰子。编写一个 Figaro 程序表现这个游戏。

a）计算滚动 12 面骰子的概率。

b）计算掷出数字 7 的概率。

c）已知掷出的是 7，计算滚动的是 12 面骰子的概率。

d）已知滚动的是 12 面骰子，计算掷出数字 7 的概率。

8．现在，修改练习 7 中的游戏，轮盘有卡住的趋势，在连续两次转动时停在同一个结果处。使用与 makeStreaky 类似的逻辑，编写一个约束，说明两次相邻的转动得到相同值的概率高于不同值的概率。连续玩该游戏两次。

a）计算第二次掷骰子得到 7 的概率。

b）已知第一次掷骰子得到 7，计算第二次掷出 7 的概率。

第 3 章　创建一个概率编程应用程序

本章介绍如下内容：

- 概率编程应用常见架构的使用
- 仅用简单的语言特性设计一个概率模型
- 从数据学习模型，根据结果推断未来的情况

现在，您已经简单了解了 Figaro 的许多特性。您可以用它们做什么？如何用它们构建有用的软件？本章简单地介绍如何用 Figaro 构建一个实际应用。

本章中，您将看到基于概率编程的垃圾邮件过滤器的完整设计，包括模型设计、判断入站邮件并将其分类为正常邮件或者垃圾邮件的组件以及从电子邮件训练集学习垃圾邮件过滤模型的组件。在此过程中，您将学习到概率编程应用中常用的架构。

3.1　把握全局

您打算构建一个垃圾邮件过滤器。如何将其融入更大的电子邮件应用？图 3-1 说明了它的工作原理。想象您有一个电子邮件服务器，其工作之一是获得入站邮件并递交给用户。当入站邮件到达时，服务器将其传递给垃圾邮件过滤器的推理组件。该组件使用概率推理确定该邮件为垃圾邮件的概率。然后，将这个概率传递给一个过滤器，后者使用垃圾邮件概率，根据某种策略决定是否允许电子邮件通过。

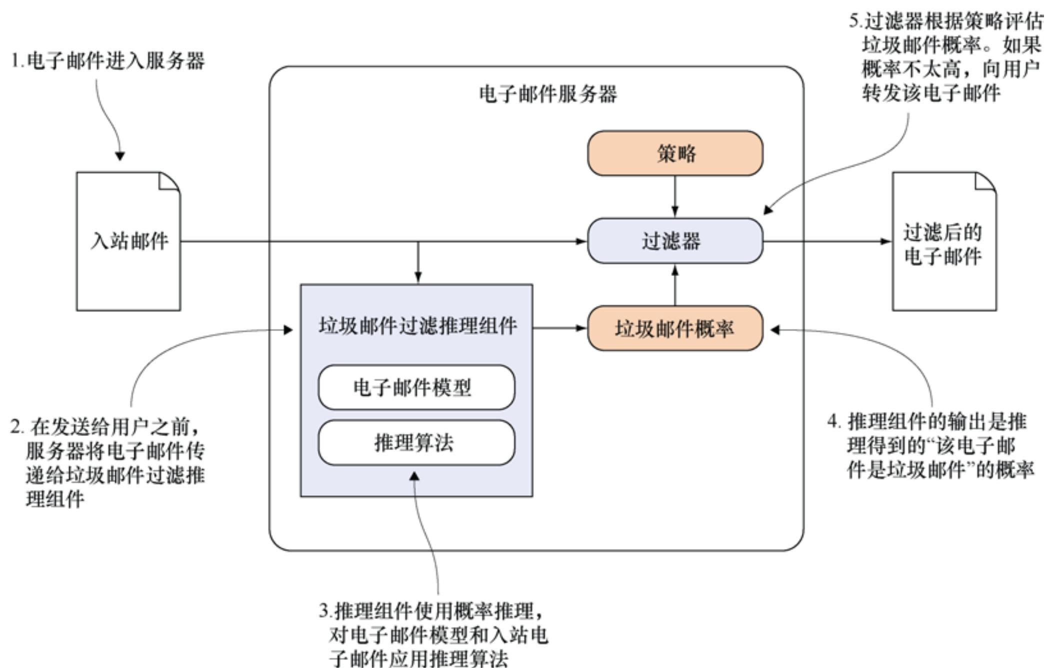


图 3-1　电子邮件服务器中垃圾邮件过滤器的运作方式。
这是每当新电子邮件到达时都会发生的联机过程

您可能记得在第 1 章中，概率推理系统使用工作于某个模型上的推理算法，根据证据回答查询。在本例中，该系统使用包含在电子邮件模型中的正常和垃圾邮件模型。查询是“电子邮件是不是垃圾邮件”，答案是垃圾邮件概率。为了正常工作，推理组件必须从某处得到电子邮件模型。这个模型从何而来？通过一个学习组件，从训练数据中“学习”。

在概率推理中，**学习**是取得训练数据并产生一个模型的过程。第 1 章简单地介绍了学习，称之为概率推理系统所能完成的工作之一。例如，足球推理系统可以使用一个赛季的角球数据学习得出一个角球模型，然后用它预测下一次角球的结果。在实际应用中，学习被当作创建用于概率推理的模型的一种方法。其他方法是使用待建模系统的知识人工设计模型。

图 3-2 展示了学习组件产生电子邮件模型的过程。这是一个离线组件；该过程不随每个电子邮件而发生，可以花费很长的时间产生最佳模型。学习组件完成其工作，产生电子邮件模型之后，推理组件才能运行。结束工作之后不再需要学习组件，除非您想要在未来的某个时间更新电子邮件模型。

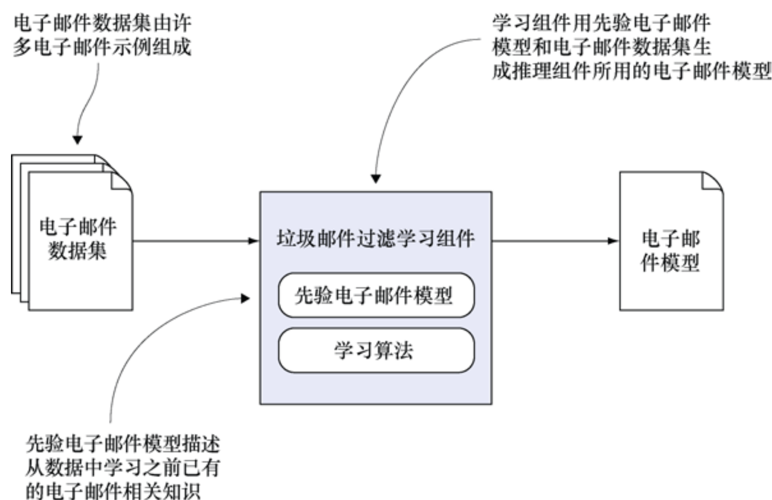


图 3-2　学习组件生成电子邮件模型的方式。学习过程可能花费很长的时间，在线下进行

学习组件使用的训练数据由一组电子邮件组成。这个数据集包含正常邮件和垃圾邮件，但是不一定都标记为正常或者垃圾。学习组件开始时并非毫无知识，而是使用在数据集上操作的学习算法，取得一个先验电子邮件模型，将其转换为推理组件使用的电子邮件模型。此时，您可能认为我只是将创建模型的工作踢给别人。如果学习组件的目标是创建电子邮件模型，那么先验电子邮件模型从何而来？

在这个例子中，先验电子邮件模型包含构建垃圾邮件过滤器所需的最小结构假设。特别是，这个模型规定电子邮件是否垃圾邮件取决于某些词语的出现或者缺失，但是不规定哪些词语代表垃圾邮件或者相关的概率。后者从数据中学习获得。先验模型规定电子邮件中不寻常单词的数量与邮件是不是垃圾邮件有某种关系，但是不规定是何种关系。同样，这种关系也从数据中学习。

稍后，我将详细解释学习组件的学习方法，简单地说，它包含如下信息。

- 哪些单词对分类正常邮件或者垃圾邮件有用。这些单词称为**特征单词**。

- 一组参数，表示观察任何给定电子邮件之前该邮件是垃圾邮件的概率、特定单词出现在正常或者垃圾邮件中的概率以及正常或者垃圾邮件包含许多不正常单词的概率等。

第一项——特征单词在第一次读取数据集时立即获得。学习组件的核心在于参数的学习。这些参数之后供推理组件使用。

人们在如何构建高效的垃圾邮件过滤器方面已经做了许多研究。本章提供的是一个相对简单的解决方案，足以说明原理，但是要组合为高效的应用程序，您还需要做很多的工作。

3.2 运行代码

在本章的主体部分中，您将看到本应用与概率编程相关的所有代码。这个应用程序包含相当数量的专用于文件 I/O 的代码，特别是从文件中读取电子邮件和将学习到的模型写入文件，再从文件中读出的代码。我不向您展示那些代码，因为它们和概率编程的主题关系不大，只会使本章的篇幅变得非常冗长。在本章看到的代码是一些片段。要运行整个应用需要完整的代码，可以在 http://manning.com/pfeffer/PPP_SourceCode.zip 找到。

学习组件需要数据。您还需要在系统开发中进行测试和评估的数据。虽然存在多个真实的垃圾邮件数据集，但是它们在互联网上的存在并不可靠，所以为了确保有可用的数据集，我已经生成了自己的数据。这不是真实电子邮件的数据集，每个“电子邮件”都表现为从电子邮件中找到的单词列表，没有试图将其连成句子或者包含标题。但是正常和垃圾邮件中单词的分布来源于真实的正常和垃圾邮件，所以从这个角度看它还是实际的数据。对于解释本章中概念的目的来说，这个数据集足够了。对于实际应用，您应该使用合适的电子邮件。

我的数据集在名为 Chapter3Data 的目录中，包含两个子目录：Training 目录包含用于训练的 100 个电子邮件示例，其中 60 个是正常邮件，40 个是垃圾邮件；Test 包含用于测试学习结果的 100 个邮件。此外，Labels.txt 文件包含所有电子邮件的标签（1 表示垃圾邮件，0 表示正常邮件）。学习组件使用这些标签帮助模型学习。推理模型不使用这些标签，那样做就是作弊了。但是当您想要评估模型时，将使用标签检查推理组件是否得到正确的答案。原则上，可能没有为所有训练邮件提供标签。即使只标记电子邮件的一个子集，学习也是有效的，本章中的代码只使用 1 个标记过的子集。但是，为了简单起见，数据集中的所有电子邮件都有标签。

在构建学习应用时，需要一种在开发期间评估应用的手段。存储库中的代码包含了一个评估程序，其任务是评估学习到的模型的质量。因为这不是部署应用的一部分，所

以我不在正文中描述它。评估程序的操作很简单，按照本文中的解释，您应该能够理解 Evaluator.scala 中的代码。

运行本章中代码的最简单方式是使用 Scala 构建工具 sbt。首先，您将在某些训练数据上运行学习组件。学习组件取 3 个参数：包含训练电子邮件的目录路径、包含标签的文件路径以及保存学习结果的文件路径。例如，您可以调用：

```
sbt "runMain chap03.LearningComponent Chapter3Data/Training Chapter3Data/
Labels.txt Chapter3Data/LearnedModel.txt"
```

产生的输出如下。（这里只展示了最相关的几行，而不是完整的结果）

```
Number of elements: 31005
Training time: 4876.629
```

结果显示创建了 31005 个 Figaro 元素，学习花费了 4876 秒。

运行学习组件最重要的结果是项目目录中出现了一个名为 LearnedModel.txt 的文件。这是一个文本文件，包含从学习到支持推理所需的信息。特别是，它包含了用于分类的所有相关单词列表以及模型的所有参数值。这个文件的格式特定于垃圾邮件过滤应用，推理组件知道如何读取。

注意：学习组件使用许多内存。您可能发现自己必须明确地设定 Java 虚拟机使用的堆空间。可以在 Java 初始化时包含-Xmx8096m 之类的选项来实现。在 Eclipse 中，可以在 Eclipse 安装目录中的 eclipse.ini 加入这个选项。在这种情况下，该选项告诉 Java 分配 8096 MB（8GB）堆空间。

有了这个模型之后，可以使用推理模型分类特定电子邮件是不是垃圾邮件。为此，运行推理组件，该组件以想要分类的电子邮件路径和学习模型路径为参数。例如，可以这样调用：

```
sbt "runMain chap03.ReasoningComponent Chapter3Data/Test/TestEmail_3.txt
Chapter3Data/LearnedModel.txt"
```

上述调用输出测试电子邮件 3 是垃圾邮件的概率。

最后，可以在测试集上评估学习结果。您可以调用评估程序完成这项工作，该程序有 4 个参数。前三个参数是包含测试电子邮件的目录路径、包含标签的文件路径以及包含学习模型的文件路径。最后一个参数对应于图 3-1 中的策略。该参数提供将电子邮件归类为垃圾邮件的概率阈值。如果推理组件认为电子邮件是垃圾邮件的概率高于阈值，就将其归类为垃圾邮件；否则归类为正常邮件。这个阈值控制垃圾邮件过滤器的敏感度。如果将阈值设置为 99%，过滤器将仅仅拦截很确定是垃圾邮件的邮件，但是许多垃圾邮件可能通过。如果阈值为 50%，可能拦截大量的非垃圾邮件。例如，运行如下命令：

```
sbt "runMain chap03.Evaluator Chapter3Data/Test Chapter3Data/Labels.txt
Chapter3Data/LearnedModel.txt 0.5"
```

将产生如下的输出：

```
True positives: 44
False negatives: 2
False positives: 0
True negatives: 54
Threshold: 0.5
Accuracy: 0.98
Precision: 1.0
Recall: 0.9565217391304348
```

上述结果总结了垃圾邮件过滤器的性能。**true positives（真阳性**）说明被过滤器分类为垃圾邮件的邮件确实是垃圾邮件的个例数量。**false positives（假阳性）**表示正常邮件被分类为垃圾邮件的数量。**false negatives（假阴性）**正相反：垃圾邮件被分类为正常邮件。最后，**true negatives（真阴性）**是正确地归类为正常邮件的邮件数量。下一行显示分类使用的阈值。最后三行显示各种性能指标。分类程序性能计量的主题很有趣，但是总体来说对概率编程并不特别重要，所以我在补充材料“计量分类程序性能”中提供细节。

本章中的代码在包含 200 个电子邮件的训练集上运行相当快。运用第 12 章中讨论的高级方法，您可以使用 Figaro 从更大的数据集中学习。

计量分类程序性能

分类程序性能的量化可能很困难。最明显的计量指标是精确度，也就是正确分类的比例。但是如果您感兴趣的大部分情况都是阴性，而您所感兴趣的是找出阳性，那么这一指标可能产生误导。例如，假定 99%的情况是阴性。只需要将任何东西都归类为阴性，就能获得 99%的精确度，但是在现实中，您得到的是一个很差的分类程序，因为它永远不能找出您感兴趣的情况。因此，人们常常使用准确率和召回率等指标。**准确率**是分类程序避免假阳性的能力。**召回率**计量其发现阳性结果的能力（避免假阴性的能力）。

精确的定义如下。*TP* 表示真阳性数量，*FP* 表示假阳性数量，*FN* 是假阴性数量，*TN* 是真阴性数量。

$$\text{精确度}=\frac{TP+TN}{TP+FP+FN+TN}$$

$$\text{准确率}=\frac{TP}{TP+FP}$$

$$\text{召回率}=\frac{TP}{TP+FN}$$

现在您已经理解了如何训练和运行垃圾邮件过滤器，我们来看看它工作的情况。首先，从应用程序的整体架构开始，然后学习模型细节和代码。

3.3　探索垃圾邮件过滤应用的架构

回到 3.1 节，我们的垃圾邮件过滤器包括两个组件。一个在线组件执行概率推理，将电子邮件分类为正常邮件或者垃圾邮件，正确地过滤或者传递给用户。一个离线组件从电子邮件训练集中学习电子邮件模型。下面几个小节描述推理组件和学习组件的架构。您将会发现，它们有许多共同部分。

3.3.1　推理组件架构

在决定组件的架构时，需要首先确定输入、输出及两者之间的关系。对于我们的垃圾邮件过滤应用，您的目标是以电子邮件为输入，确定它是正常邮件还是垃圾邮件。因为这是一个概率编程应用，所以不会以布尔型的垃圾/正常邮件分类为输出，而是产生电子邮件是垃圾邮件的概率。为了简单起见，您将假定电子邮件输入是一个文本文件。

垃圾邮件过滤推理组件总结如下。

- **输入**——表示电子邮件的文本文件。
- **输出**——表示电子邮件是垃圾邮件的概率的双精度类型值。

现在，您将一步一步地构建垃圾邮件过滤器的架构。

第 1 步：以概率推理系统的形式定义概要架构

首先，您可以回到第 1 章中介绍的概率推理系统基本架构，粗略地制定垃圾过滤应用的架构，如图 3-3 所示。**垃圾邮件过滤推理组件**用于确定电子邮件是正常邮件还是垃圾邮件。它以电子邮件文本作为证据。查询是“该邮件是不是垃圾邮件”。答案是电子邮件是垃圾邮件的概率。

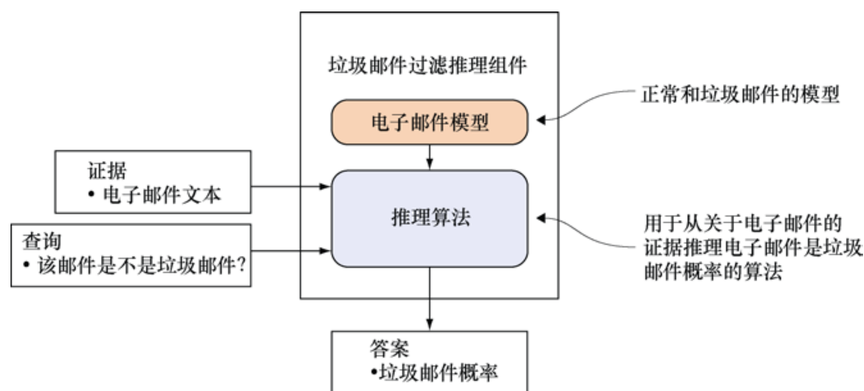


图 3-3　垃圾邮件过滤推理架构概要。垃圾过滤器使用在电子邮件模型上操作的推理算法，根据邮件文本确定该邮件是垃圾邮件的概率

为了获得这个答案，推理组件使用**电子邮件模型**和**推理算法**。电子邮件模型是捕捉关于电子邮件一般知识的概率模型，这些知识包括电子邮件的属性和该邮件是否垃圾邮件。推理算法使用该模型，根据证据回答“电子邮件是不是垃圾邮件”的查询。

第 2 步：完善垃圾邮件过滤应用的架构

您想要使用概率程序表示概率模型。在 Figaro 中，概率程序中包含元素。回顾第 2 章，在 Figaro 中，证据以条件或者约束的形式应用到单独元素。所以，您需要一个将电子邮件文本转换为可应用到模型单独元素的证据的手段。在机器学习中，将原始数据转化为关于模型元素证据的组件一般称作**特征提取程序**（**feature extractor**）。数据中应用到模型上作为证据的方面被称作**特征**。如图 3-4 所示，特征提取程序以电子邮件文本作为输入，将其转换为一组电子邮件特征，这些特征成为推理算法的证据。

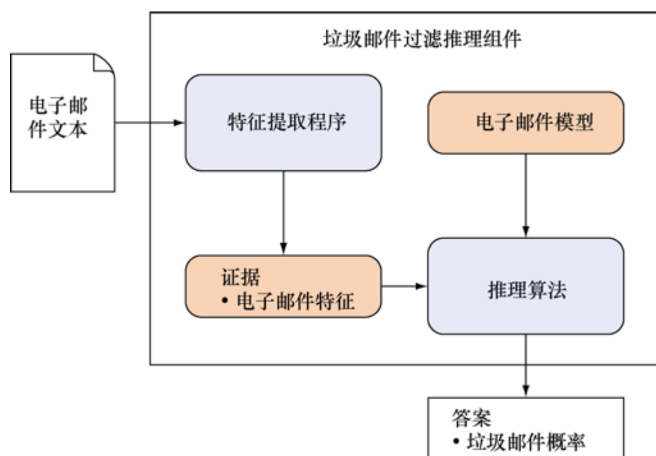


图 3-4 垃圾邮件过滤推理架构的第二稿。您已经添加了特征提取程序，查询始终相同，因此删除

而且，这个特殊的概率推理应用始终回答同一个查询：该电子邮件是不是垃圾邮件？您不需要将查询作为应用的输入，因此从架构中删去查询。

第 3 步：详细描述架构

现在，是时候仔细观察模型了。图 3-5 展示了完善之后的推理架构，它将模型分为三部分：过程，包括模型设计人员编程的结构化知识；从数据学习而来的参数；与元素不直接相关但是对推理有用的辅助知识。

创建模型需要技能，本书包含许多例子，但是一般来说，概率模型中的组成部分都相同。我将在 3.4 节中详细介绍垃圾邮件过滤应用的这些成分，但是在这里也将做简单的介绍。该模型包含 5 个不同的成分。

- **定义模型中各个元素的模板**。例如，一个元素表示电子邮件是不是垃圾邮件，其他元素表示电子邮件中是否存在特定的单词。这里没有指定具体的单词；那是知识的一部分，在这个列表的最后解释。3.4.1 小节讨论垃圾过滤器模型元素的选择。
- **元素之间的相互依赖关系**。例如，表示特定单词存在的元素依赖于表示电子邮件是垃圾邮件的元素。记住，在概率模型中，依赖关系的方向不一定是推理的方向。您使用单词确定电子邮件是否垃圾邮件，但是在模型中，您可以认为电子邮件发送方首先决定发送的邮件类型（“我打算向你发送一封垃圾邮件”），然后决定邮件中的单词。通常按照因果的方向建立依赖关系的模型。因为，电子邮件的种类造成用词的不同，依赖关系从表示电子邮件是不是垃圾邮件的元素指向每个单词。垃圾邮件过滤器模型中依赖关系的确定在 3.4.2 小节中详加讨论。

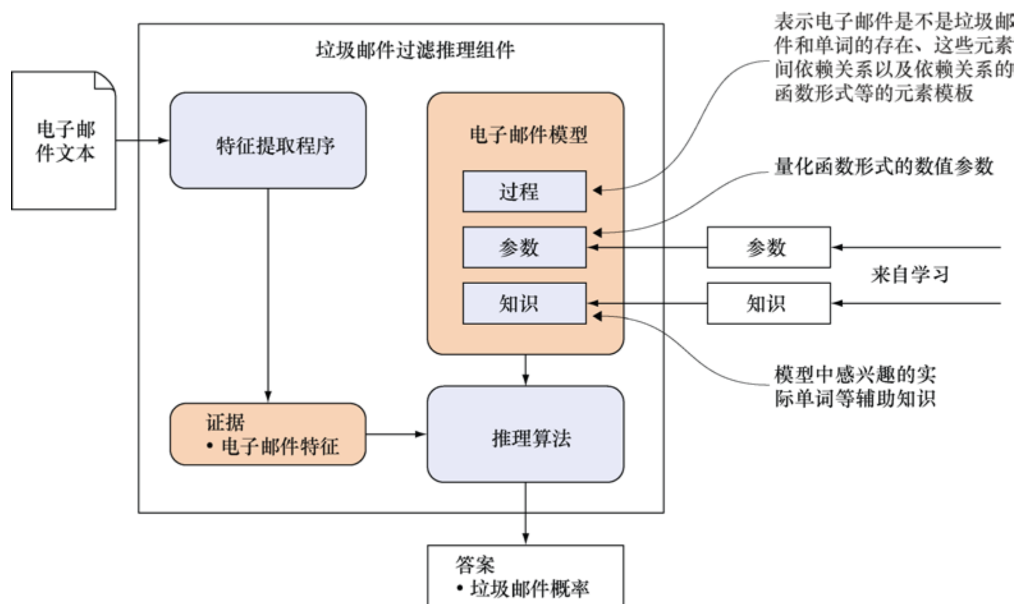


图 3-5 垃圾邮件过滤推理架构的第三稿。该模型被分解为过程、参数和知识。参数和知识来自学习

- **这些依赖关系的函数形式**。元素的函数形式表明了元素的类型（如 If、原子 Binomial 或者复合 Normal）。函数形式没有指定元素的参数。例如，前一个依赖关系采用包含两个 Flip 的 If 形式：如果电子邮件是垃圾邮件，则有某种概率出现单词 rich；否则，该单词以另一种概率出现。垃圾邮件过滤器模型的函数形式在 3.4.3 小节中定义。

前 3 个成分组成了图 3-5 中的过程，是垃圾邮件过滤应用在学习之前已知的。它们由应用设计者定义，一起组成电子邮件模型的结构。

- **模型的数值参数**，在图 3-5 中显示为“参数”。例如，某个电子邮件是垃圾邮件的概率是参数之一。某个邮件是垃圾邮件时，单词 rich 出现的概率是另一个参数；某个邮件不是垃圾邮件时，单词 rich 出现的概率是第三个参数。这些参数与过程分离，因为参数将从训练数据中学习。电子邮件模型的参数将在 3.4.4 小节中讨论。
- **知识**。图 3-5 展示了用于构造模型，并在特定情况下向其应用证据所用的辅助知识。我用“知识”表示从训练数据中学习到的任何不与元素、元素间依赖关系、函数形式或者数值参数相关的信息。在我们的垃圾邮件过滤器应用中，这些知识由包含训练电子邮件中出现的单词以及每个单词出现次数的字典组成。例如，您需要这些知识告诉您，单词“rich”是您感兴趣的。因此，知识帮助定义模型的参数；只有在 rich 是您所感兴趣的单词时，电子邮件是垃圾邮件时 rich 出现的概率才是一个参数。辅助知识将在 3.4.5 小节进一步讨论。

现在，推理组件的架构已经相当完整，可以研究学习组件了。

3.3.2 学习组件架构

从 3.1 小节中，您知道学习组件的任务是根据一个电子邮件数据集，产生推理组件使用的电子邮件模型。正如推理组件中那样，第一件要做的事情是确定学习组件的输入和输出。从图 3-5 中，您知道输出必须是电子邮件模型的参数和知识。但是，输入是什么呢？您可以假定自己得到了一个用于学习的电子邮件训练集，由它组成输入的一部分。但是如果有人已经将这些电子邮件标记为正常邮件或者垃圾邮件，学习就轻松多了。您将假定自己得到了一个电子邮件训练集，以及其中一些邮件的标记。但是，并不是所有电子邮件都必须标记，您可以使用大的未标记电子邮件数据集和较小的带标记数据集。

下面是垃圾过滤学习组件的总结。

- **输入**
  - —— 一组代表电子邮件的文本文件。
  - —— 包含电子邮件子集的正常/垃圾邮件标签的文件。
- **输出**
  - —— 描述生成过程特征的数值参数。
  - —— 模型中使用的辅助知识。

图 3-6 展示了垃圾邮件过滤学习组件的架构。它使用了许多与推理组件类似的元素，但是有一些重要的差异。

- 学习组件以**先验电子邮件模型**为中心。这个模型包含的组成部分与推理组件中的电子邮件模型相同，但是使用**先验参数**而非学习组件产生的参数。在第 5 章中您将看到，先验参数是模型在观察任何数据之前的参数。因为学习组件在操作之前没有观察任何数据，所以使用先验参数值。模型的其他两部分——**过程**和**知识**——与推理组件中完全相同。

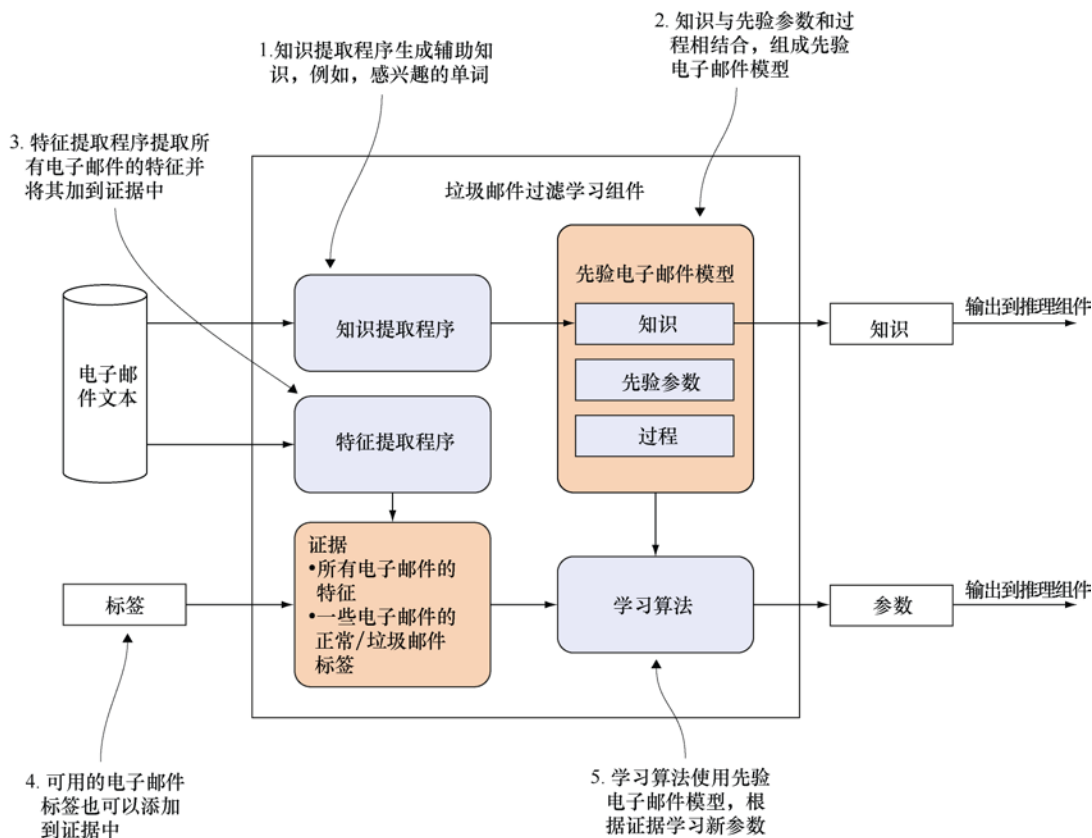


图 3-6　学习组件架构

- **知识提取程序**直接从训练电子邮件提取知识。例如，知识可能包含出现在电子邮件中的最常见单词。这一知识还将从学习组件输出、发送给推理组件。
- 正如推理组件中那样，**特征提取程序**提取所有电子邮件的特征，并将其转换为证据。对于有标签的电子邮件，正常/垃圾邮件标签也被转换为证据。

- 学习组件中用**学习算法**替代推理算法。这个算法使用关于所有电子邮件的所有证据，使用模型学习参数值。和推理算法类似，Figaro 提供学习算法，稍后您将在本书中学习到该算法。

上面介绍的都是学习组件的组成部分。将架构分割为推理和学习组件的好处之一是，学习组件可以运行一次，其结果可以供推理组件多次使用。从训练数据中学习是费时的操作，您不应该在每次推断新电子邮件时都运行它。您的设计可以进行一次学习，输出学习到的参数和知识，用它们快速地推断入站电子邮件。

现在您已经了解了架构，下面可以观察实践中的工作方式。下一节介绍模型的设计和为每个电子邮件构建模型的代码。然后，您将看到推理组件的实现，最后是学习组件的实现。

3.4　设计电子邮件模型

本节介绍如何设计正常和垃圾邮件的概率模型。因为推理和学习组件使用相同的过程和知识，差别仅在于参数，因此设计对两者都适用。

在 Figaro 中构建概率模型时，需要组合 4 种成分。

- 模型中的元素。
- 元素相互连接的方式（元素间的依赖关系）。
- 依赖关系的函数形式。这些函数形式是用于实现依赖关系的元素类构造程序。例如，原子 Flip 是不依赖于任何其他元素的布尔（Boolean）元素的元素类构造程序，而复合 Flip 是依赖于另一个元素决定自身含义的双精度元素构造程序。
- 函数形式的数值参数。

我们将依次考察这些成分。最后，我还将描述模型中的辅助知识。

3.4.1　选择元素

在设计垃圾邮件过滤器时，必须考虑关于电子邮件的哪些情况能够说明它是垃圾邮件还是正常邮件，其中一些情况包括：

- 电子邮件标题或者正文中有特定的单词。例如，rich 在垃圾邮件中出现的可能性高于在正常电子邮件中出现的可能性。
- 错误的拼写和其他不正常的单词。垃圾邮件往往包含错误拼写的词或不出现在其他电子邮件中的单词。
- 特定单词出现在其他单词附近。
- 自然语言特征，如词性。
- 标题的特征，如发信人和收信人地址、使用的服务器等。

为了保持应用程序相对简洁，您将做出简化的设计决策。

- 只使用前两个项目：电子邮件中的单词以及不寻常单词的出现。
- 尽管单独对电子邮件标题和正文进行推理很有好处，但是您将把两者混在一起处理。
- 为电子邮件中的某个单词创建元素时有两种选择。一是使用整数元素，其值是电子邮件中出现该单词的次数。二是使用布尔元素，其值表示该单词是否出现，而不管其出现次数。您将使用第二种选择。
- 对于不寻常的单词，您可以在语言词典中查找每个单词是正确定义的单词还是不平常的单词。为了避免依赖外部应用，您将使用不同的方法，例如，不寻常的单词是不出现在任何其他电子邮件中的单词。

根据这些假设，您的下一个任务是选择模型的元素。您将有 3 组元素。第一组表示电子邮件是不是垃圾邮件。这是通过一个布尔元素实现的。第二组包括电子邮件中出现或者缺少特定单词。第三组与电子邮件中不寻常单词的数量有关。

表示单词是否出现

创建表示单词是否出现的元素时，必须思考模型中包含了哪些单词。您可能对出现在任何训练邮件中的每个单词建立一个元素。但是这将造成元素数量很大。例如，包含 1000 个电子邮件的典型训练集包含 40000 多个不同的单词。

元素太多很不好，原因有二。首先，这会造成**过度拟合**，也就是说您将学习到一个能够很好地拟合训练数据的模型，但是却不能很好地概括。当您的元素数量很大时，其中一些可能对训练数据做出不合逻辑的解释，而实际上它们并不能预测某个邮件是不是垃圾邮件。想象一下，您在 5 个电子邮件中随机地放入一个单词。假定训练电子邮件中有 1/3 是垃圾邮件。出现该单词的 5 个电子邮件都是垃圾邮件的概率大约为 1/250。这也就意味着，如果您有 40000 个单词出现了 5 次，其中只有大约 160 个单词出现在垃圾邮件中。这 160 个单词可以解释所有垃圾邮件。但是根据这个实验性的假设，这些单词是随机出现在训练电子邮件中的，所以它们没有预测能力。

太多元素不好的另一个理由是这会使得推理和学习很慢。您希望推理尽可能快，因为需要将其应用到进入服务器的每封电子邮件上。与此同时，学习概率模型可能是一个缓慢的过程，特别是在拥有数千个（或者更多）训练实例的情况下。使用大量元素可能使其慢到无法使用的地步。

根据这些考虑因素，您将在模型中为 100 个单词创建元素。下一个问题是，哪 100 个单词？可以使用称为**特征选择**的机器学习技术决定模型中包含的特征或者元素。在垃圾过滤器中，特征选择的目标是识别最可能表示电子邮件是正常还是垃圾邮件的单词。特征选择技术可能十分复杂。如果您打算构建一个生产级别的垃圾邮件过滤器，可以使用这些复杂技术中的一种。但是本章将使用一种简单的技术。

这种技术基于两种考虑。首先，使用出现频率过低的单词将导致过度拟合，因为它

们的出现过于随机，没有预测性。其次，the 和 and 等频繁出现的单词也可能没有预测性，因为它们在正常和垃圾邮件中都出现很多次。这种单词称作**停用词**。所以，为了创建模型中的单词列表，您将删除所有过于频繁出现的单词，使用下 100 个最常见单词。

表示不寻常单词的数量

在模型中包含不寻常单词数量的动机是垃圾邮件倾向于包含比常规邮件更多的不寻常单词。实际上，在包含 1000 封电子邮件的训练集中，垃圾邮件中有 28%的单词是不寻常的，而常规邮件中只有 18%的单词是不寻常的。但是对数据进行更仔细的观察，就会发现更微妙的情况。有些垃圾邮件有许多不寻常单词，而有些垃圾邮件在不寻常单词的数量方面和正常电子邮件类似。为了捕捉这一细节，您在模型中添加 hasUnusualWords 元素，表示电子邮件中是否包含许多不寻常单词。这是一个**隐含元素**，它不直接出现在数据中，但是在模型中包含它可以大大提高精确度。

总结一下，电子邮件模型中的元素如下。

- isSpam——Boolean 元素，如果电子邮件是垃圾邮件，其值为 true。
- hasWord1，…，hasWord100——对应每个单词的布尔元素，如果该单词在电子邮件中出现，值为 true。
- hasManyUnusualWords——布尔元素，如果电子邮件有许多不寻常的单词，值为 true。
- numUnusualWords——整数型（Integer）元素，值为邮件中没有出现在任何其他电子邮件的单词数量。

您可以用下面的 Model 类总结模型中的元素。这个抽象 Model 类声明了模型中的元素，但是没有提供其定义。创建抽象 Model 类的目的是用 ReasoningModel 和 LearningModel 进行扩展。后面您将看到，由于推理模型和学习模型之间的细微差别，有必要创建两个类，使用抽象基类意味着可以编写适用于任何一类模型的方法（如应用证据）。在第 12 章中，您将学习一种模式，该模式使用单一模型类创建推理和学习组件，从而免去创建两个类的麻烦，但是它使用了您尚未学习到的 Figaro 特性。

程序清单 3-1　抽象 Model 类

```
abstract class Model(val dictionary: Dictionary) {
  val isSpam: Element[Boolean]
  val hasManyUnusualWords: Element[Boolean]
  val numUnusualWords: Element[Int]

  val hasWordElements: List[(String, Element[Boolean])]
}
```

模型类以一个字典作为参数。在 3.4.5 小节中您将看到，该字典包含关于训练集中单词的辅助知识

模型中元素的声明，和前几段的描述一致

hasWordElements 是（单词，元素）配对，每个元素表示对应的单词是否出现在电子邮件中

3.4.2　定义依赖关系

我们首先定义 isSpam 元素和所有 word 元素的依赖性模型。定义概率模型的依赖关系时，常用的经验法则是对象类决定对象属性，也就是说属性取决于依赖性模型中的类。在我们的例子中，电子邮件是不是垃圾邮件决定了所有单词是否出现。

前面我已经提到过这一点，但是值得重申，因为它很重要，人们往往对此感到困惑。您将使用单词确定电子邮件是不是垃圾邮件。那么，为什么我说电子邮件的分类决定单词，而不是反过来呢？关键的要点是，在概率推理中，推理的方向不一定是模型中依赖性的方向。推理的方向往往和依赖性的方向相反，因为您试图确定导致观测结果的因素。这里，您建立的是电子邮件本质（是垃圾邮件还是正常邮件）的模型，电子邮件的本质是没有观察到的因素，它是观测到的单词的根源。所以，模型中的依赖关系方向是从电子邮件分类到单词。

话虽如此，您还需要确定是否建立单词之间依赖性的模型。在英语或者任何语言中，句子第一个单词的选择与第二个单词紧密相关。实际上，单词不是独立的。但是为了保持本章的简洁，我假设在电子邮件分类（正常或者垃圾）给定的情况下，单词是独立的。

您可以绘制名为**贝叶斯网络**的框图，说明模型中的依赖关系。第 5 章将介绍贝叶斯网络。现在需要了解的一点是，如果网络中的一个节点依赖于另一个节点，则两个节点之间存在一条边。对于目前为止的垃圾邮件过滤器，可以得到图 3-7 中所示的贝叶斯网络。这种网络称为**朴素贝叶斯模型**——称之为朴素，是因为它假设单词是相互独立的。

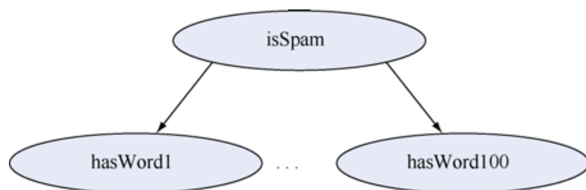


图 3-7　isSpam 和 word 元素的朴素贝叶斯依赖性模型

下面，我们介绍用于不寻常单词的元素。记住，您有两个元素：hasManyUnusualWords 和 numUnusualWords。hasManyUnusualWords 表示电子邮件中的单词是否倾向于不寻常单词，而 numUnusualWords 是一个数字。因为包含大量不寻常单词的电子邮件的不寻常单词数量很可能比不包含大量不寻常单词的电子邮件大，numUnusualWords 自然依赖于 hasManyUnusualWords。而 hasManyUnusualWords 是电子邮件的属性，依赖于表示电子邮件分类的 isSpam。最后，继续朴素贝叶斯模型的假设，在分类已知的情况下，不寻常单词的出现独立于任何给定特殊单词的出现。您最终得到图 3-8 中的贝叶斯网络。这是垃圾邮件过滤器应用的最终依赖性模型。

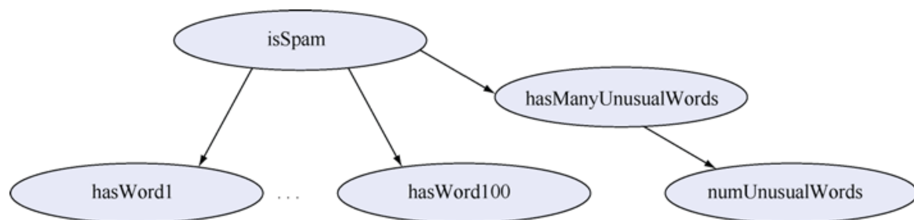


图 3-8　包含不寻常单词的完整依赖性模型

3.4.3　定义函数形式

您必须为 4 组元素定义函数形式。函数形式同样是构建模型所用的元素类构造程序。除了未指定的数值之外，表达模型所需的就是它们了。我将依次介绍这些函数形式。首先，为推理组件定义函数形式，在这个组件中，假定学习组件已经生成了参数值。下面是推理模型的代码，我将在代码之后详细解释每个元素的定义。

程序清单 3-2　推理模型

```
class ReasoningModel(
  dictionary: Dictionary,
  parameters: LearnedParameters
) extends Model(dictionary) {
  val isSpam = Flip(parameters.spamProbability)

  val hasWordElements = {
    for { word <- dictionary.featureWords } yield {
      val givenSpamProbability =
        parameters.wordGivenSpamProbabilities(word)
      val givenNormalProbability =
        parameters.wordGivenNormalProbabilities(word)
      val hasWordIfSpam = Flip(givenSpamProbability)
      val hasWordIfNormal = Flip(givenNormalProbability)
      val hasWord = If(isSpam, hasWordIfSpam, hasWordIfNormal)
       (word, hasWord)
    }
  }

  val hasManyUnusualIfSpam =
    Flip(parameters.hasManyUnusualWordsGivenSpamProbability)
  val hasManyUnusualIfNormal =
    Flip(parameters.hasManyUnusualWordsGivenNormalProbability)
  val hasManyUnusualWords =
    If(isSpam, hasManyUnusualIfSpam, hasManyUnusualIfNormal)
```

推理模型的参数是学习产生的字典和参数

ReasoningModel 实现抽象 Model 类

表示电子邮件是不是垃圾邮件的元素

表示某个单词是否出现在电子邮件中的元素。这段代码创建一个（单词，元素）配对列表，每个列表项代表一个特征单词

表示电子邮件中是否有许多不寻常单词的元素

```
  val numUnusualIfHasMany =
    Binomial(Model.binomialNumTrials,
             parameters.unusualWordGivenManyProbability),
  val numUnusualIfHasFew =
    Binomial(Model.binomialNumTrials,
             parameters.unusualWordGivenFewProbability),
  val numUnusualWords =
    If(hasManyUnusualWords, numUnusualIfHasMany, numUnusualIfHasFew)
}
```

表示不寻常单词数量的元素

现在，我们来详细了解一下模型中的每个元素。

- isSpam——这个布尔元素为 true 的概率就是给定电子邮件是垃圾邮件的概率。所以使用 Flip 元素定义它。在 Figaro 中，定义为 val isSpam = Flip(parameter.spamProbability)。这反映您在查看任何单词之前，电子邮件是垃圾邮件的概率。因为单词取决于电子邮件是不是垃圾邮件，而且单词可以观察到，所以从单词推断电子邮件是不是垃圾邮件。这就是依赖性的方向与推理方向不同的原因。在概率推理中，您可以从某一根源产生的结果倒推出根源。
- word1， …， word100——每个单词的出现都有某种概率。这个概率取决于电子邮件是正常邮件还是垃圾邮件。对此 If 结构最为合适，基本形式为：

```
val hasWordIfSpam = Flip(givenSpamProbability)
val hasWordIfNormal = Flip(givenNormalProbability)
val hasWord = If(isSpam, hasWordIfSpam, hasWordIfNormal)
```

警告：如果您观察 hasWord 的定义，可能认为通过消除 hasWordIfSpam 和 hasWordIfNormal，编写如下简化代码是可行的:

```
val hasWord = If(isSpam, Flip(givenSpamProbability),
Flip(givenNormalProbability))
```

遗憾的是，Figaro 有一个限制，依赖于学习所得参数（如 Flip(givenSpamProbability)取决于学习得到的参数 givenSpamProbability)的元素不能在 Chain 中定义。因为 If 是 Chain 的一个种类，取决于学习所得参数的元素不能在 If 中定义。我们希望在未来的 Figaro 版本中消除这个限制。

- givenSpamProbability 和 givenNormalProbability 得自学习结果，对每个单词都不同。您将为作为特征使用的每个单词创建这样的 If 形式。在 Scala 中，创建一个列表，其中每个单词与其 hasWord 元素配对。下面是定义代码：

```
val hasWordElements = {
  for { word <- dictionary.featureWords } yield {
    val givenSpamProbability =
      parameters.wordGivenSpamProbabilities(word)
    val givenNormalProbability =
      parameters.wordGivenNormalProbabilities(word)
    val hasWordIfSpam = Flip(givenSpamProbability)
    val hasWordIfNormal = Flip(givenNormalProbability)
    val hasWord = If(isSpam, hasWordIfSpam, hasWordIfNormal)
     (word, hasWord)
  }
}
```

dictionary.featureWords 生成作为特征的 100 个单词列表。这个循环遍历所有单词

根据电子邮件是垃圾邮件还是正常邮件，单词出现的概率

创建表示单词是否出现的 Figaro 元素

单词及其元素的配对。所有这些配对用 for 循环在列表中连接起来

- hasManyUnusualWords——这个元素为 true 的概率取决于电子邮件是垃圾邮件还是正常邮件，同样，If 是合适的形式：

```
val hasManyUnusualIfSpam =
  Flip(parameters.hasManyUnusualWordsGivenSpamProbability)
val hasManyUnusualIfNormal =
  Flip(parameters.hasManyUnusualWordsGivenNormalProbability)
val hasManyUnusualWords =
  If(isSpam, hasManyUnusualIfSpam, hasManyUnusualIfNormal)
```

- numUnusualWords——这个整数型元素表示电子邮件中不寻常单词的数量。电子邮件中的单词越多，就越可能有较多的不寻常单词。您将为每个有可能的不寻常单词建立模型。表现这种情况的自然途径是使用二项分布，其中试验次数等于电子邮件中的单词总数。但是电子邮件中的单词数量可能超过 1000，当试验次数那么大时，许多 numUnusualWords 的值概率极低。所以，电子邮件的概率可能极小。这可能导致在电子邮件很多时发生下溢：概率因为取值太小而无法在计算机上表示，被舍入为 0。

  为了解决这个问题，您将试验次数设置为固定数，如 20。这个数字在代码中用 Model.binomialNumTrials 常量表示。然后，电子邮件中的不寻常单词数按比例缩小，使其表现为 20 的一部分。您将在后面应用证据时进行这项处理。使用这种方法，numUnusualWords 的定义如下：

```
val numUnusualIfHasMany =
  Binomial(Model.binomialNumTrials,
           parameters.unusualWordGivenManyProbability),
val numUnusualIfHasFew =
  Binomial(Model.binomialNumTrials,
```

```
          parameters.unusualWordGivenFewProbability),
val numUnusualWords =
  If(hasManyUnusualWords, numUnusualIfHasMany, numUnusualIfHasFew)
```

推理模型就是这样。除了不使用来自学习结果的特定概率而使用代表先验概率的元素之外，学习模型与此完全相同。例如：

```
val isSpam = Flip(parameters.spamProbability)
```

将此与前面为推理组件定义的元素做比较：

```
val isSpam = Flip(parameters.spamProbability)
```

两者看上去完全一样，但是存在显著的不同。在推理模型中，parameters.spamProbability 是固定数。而在学习模型中，parameters.spamProbability 是一个元素；它表示可以取许多可能值之一的随机元素。当您学习时，并不知道特定电子邮件是垃圾邮件，所以用一个随机元素表示概率。学习结果是这一概率的特定值，用于推理。学习模型的代码如下：

```
class LearningModel(
  dictionary: Dictionary,
  parameters: PriorParameters
) extends Model(dictionary) {
  // body is exactly the same as for the reasoning model
}
```

表示先验参数的元素

学习模型也扩展抽象 Model 类

3.4.4　使用数值参数

上面的话题将我们引入了数值参数的讨论。在推理组件中，这些参数由学习结果提供。LearnedParameters 定义模型所用的所有参数，代码清单如下。

程序清单 3-3　LearnedParameters 类

```
class LearnedParameters(
  val spamProbability: Double,
  val hasManyUnusualWordsGivenSpamProbability: Double,
  val hasManyUnusualWordsGivenNormalProbability: Double,
  val unusualWordGivenManyProbability: Double,
  val unusualWordGivenFewProbability: Double,
  val wordGivenSpamProbabilities: Map[String, Double],
  val wordGivenNormalProbabilities: Map[String, Double]
)
```

对于学习组件，您需要指定先验参数值。我们的模型使用复合 Flip 和 Binomial 元素。第 2 章中曾经介绍过，它们都以一个 Element[Double]为参数。这个 Element[Double]是复合 Flip 和 Binomial 的参数。您通常希望这个参数是一个连续原子元素。有一类连续原子元素称作 Beta 元素，很适合于作为复合 Flip 或者 Binomial 的参数，所以您将使用它。第 4 章和第 5 章中将学习到更多 Beta 元素的有关知识。

Beta 原子元素以两个双精度值为参数。在下面的代码中，这些参数的值经过精心选择，

但是您必须等到后续的章节才能理解使用这些特殊值的原因。下面是 PriorParameters 类。

程序清单 3-4 PriorParameters 类

```
class PriorParameters(dictionary: Dictionary) {
  val spamProbability = Beta(2,3)

  val wordGivenSpamProbabilities =
    dictionary.featureWords.map(word => (word, Beta(2,2)))
  val wordGivenNormalProbabilities =
    dictionary.featureWords.map(word => (word, Beta(2,2)))

  val hasManyUnusualWordsGivenSpamProbability = Beta(2,2)
  val hasManyUnusualWordsGivenNormalProbability = Beta(2, 21)
  val unusualWordGivenManyProbability = Beta(2,2)
  val unusualWordGivenFewProbability = Beta(2,7)

  val fullParameterList =
    spamProbability ::
    hasManyUnusualWordsGivenSpamProbability ::
    hasManyUnusualWordsGivenNormalProbability ::
    unusualWordGivenManyProbability ::
    unusualWordGivenFewProbability ::
    wordGivenSpamProbabilities.map(pair => pair._2) :::
    wordGivenNormalProbabilities.map(pair => pair._2)
  }
```

垃圾邮件先验概率

某个单词出现在垃圾邮件或者正常邮件中的概率

根据指定邮件是垃圾邮件还是正常邮件，它包含许多不寻常单词的先验概率

在指定邮件是否包含许多不寻常单词的情况下，某个单词是不寻常单词的先验概率

告诉学习算法所要学习的是哪些参数。参见下面的补充资料

让我们更仔细地观察下面几行：

```
val wordGivenSpamProbabilities =
  dictionary.featureWords.map(word => (word, Beta(2,2)))
val wordGivenNormalProbabilities =
  dictionary.featureWords.map(word => (word, Beta(2,2)))
```

每个单词出现在垃圾邮件中都有一定的概率，对正常电子邮件也是如此。每个单词的先验概率是 Beta(2,2)。上述代码创建一个（单词,元素）配对列表，每一项对应一个特征单词。对于每个特征单词，这个元素是 Beta(2,2)。这里有一个重要的细节：每个单词的 Beta 元素各不相同。每个单词出现在正常或者垃圾邮件中的概率也各不相同。

Scala 注释

让我们仔细地观察定义 fullParameters 的代码。这段代码创建一个列表，包含模型中的所有参数。代码中有两处值得注意。

首先看看 wordGivenNormalProbabilities.map(pair => pair._2)。这一行遍历 wordGivenNormalProbabilities 中的每个（单词，元素）配对。对于每个配对，应用函数取得一个配对并返回第二个参数。换言之，返回配对中的元素。所以，这一行代码返回与任何特征单词在正常邮件中概率相关的元素。wordGivenSpamProbabilities.map(pair => pair._2)与此类似，但是返回的是垃圾邮件相关的概率。

值得注意的第二点是所有元素组成列表的方式。如果仔细观察定义，就会看到前 5 行各包含一个元素，而最后两行包含元素的列表。在 Scala 中，:: 运算符取得元素 x 和列表 xs，将 x 附加到 xs 的前面。同时，::: 运算符连接两个列表。所以，这段代码创建一个列表，包含所有作为模型参数的元素。

3.4.5　使用辅助知识

要为特定的电子邮件构造一个模型，就需要知道哪些单词是特征单词。您还需要知道特定单词是不是不寻常的。这些服务由数据结构 Dictionary 提供。下面是 Dictionary 类的代码。在注释中可以找到完整的解释。

程序清单 3-5　Dictionary 类

```
class Dictionary(initNumEmails: Int) {

  val counts: Map[String, Int] = Map()

  var numEmails = initNumEmails

  def addWord(word: String) {
    counts += word -> (getCount(word) + 1)
  }
  def addEmail(email: Email) {
    numEmails += 1
    for { word <- email.allWords } {
      addWord(word)
    }
  }
  object OrderByCount extends Ordering[String] {
    def compare(a: String, b: String) = getCount(b) - getCount(a)
  }
  def words = counts.keySet.toList.sorted(OrderByCount)

  def nonStopWords =
    words.dropWhile(counts(_) >=
                    numEmails * Dictionary.stopWordFraction)
  def featureWords = nonStopWords.take(Dictionary.numFeatures)

  def getCount(word: String) =
    counts.getOrElse(word, 0)

  def isUnusual(word: String, learning: Boolean) =
    if (learning) getCount(word) <= 1
    else getCount(word) <= 0
  }
```

注释：
- Dictionary 类维护用来构建它的电子邮件数量，initNumEmails 是初始计数
- 从单词到出现它们的电子邮件数量的映射
- 构建这个 Dictionary 所用的邮件数量
- 递增 Dictionary 计数，为其添加一个单词
- 递增 numEmails 并添加电子邮件中的每个单词，将电子邮件加入 Dictionary
- Dictionary 中的单词，按照计数排序
- 获得出现不是过于频繁的单词
- 获得剩下单词中最频繁出现的 numFeatures
- 获得出现指定单词的电子邮件数量
- 确定单词是否不寻常

下面是对上述代码的一些说明。

- Dictionary 类的参数 initNumEmails 是字典中电子邮件的初始数量。在推理组件中，电子邮件总数已经从学习组件中得知，所以 initNumEmails 被设置为电子邮件的数量。在学习组件中，电子邮件一次添加一个，所以 initNumEmails 设置为 0。
- numEmails 是字典中的电子邮件数量，从 initNumEmails 开始，每次添加电子邮件的时候递增。电子邮件只由学习组件添加到字典中。
- words 是字典中所有单词的列表，按照计数排列，从最常见到最少见。排序方法取得 Ordering 参数，该参数提供排序列表中两个元素的 compare 方法。排序由 OrderByCount 对象实现。
- 特征单词以两个步骤组成完整的单词集。在第一步中，遍历排序的单词列表，删除所有过于频繁出现的单词（停用词）。如果在电子邮件中出现的比例至少达到 Dictionary.stopWordFraction，则该单词被视为停用词。例如，如果 Dictionary.stopWordFraction 为 0.2,训练集中有 50 封电子邮件，停用词就是至少出现在 10 封电子邮件中的单词。在第二步中，从 nonStopWords 中取出前 Dictionary.numFeatures 个单词。
- getCount 使用 Scala 库中的 getOrElse 方法，getOrElse 方法在计数映射中搜索单词，如果没有找到则返回 0。
- isUnusual 有某种逻辑。如果某个单词没有出现在任何其他电子邮件中，它被视为不寻常单词。在学习中，因为该单词出现在问题中的电子邮件里，其计数<=1。在推理中，字典仅包含训练期间看到的邮件，所以不寻常的单词完全不会出现在字典中。

3.5 构建推理组件

我们再来看看图 3-9 中重现的推理组件架构。您已经看到了电子邮件模型的各个部分、生成过程、来自学习的参数以及学习期间生成的字典形式“知识”，仍然需要学习特征提取程序、证据的陈述和推理算法运行的相关知识。

您使用 Email 类提取特征和观察证据。下面是该类的代码，在注释中加以解释。完成模型上的所有工作之后，从电子邮件提取特征和观察证据都很简单。

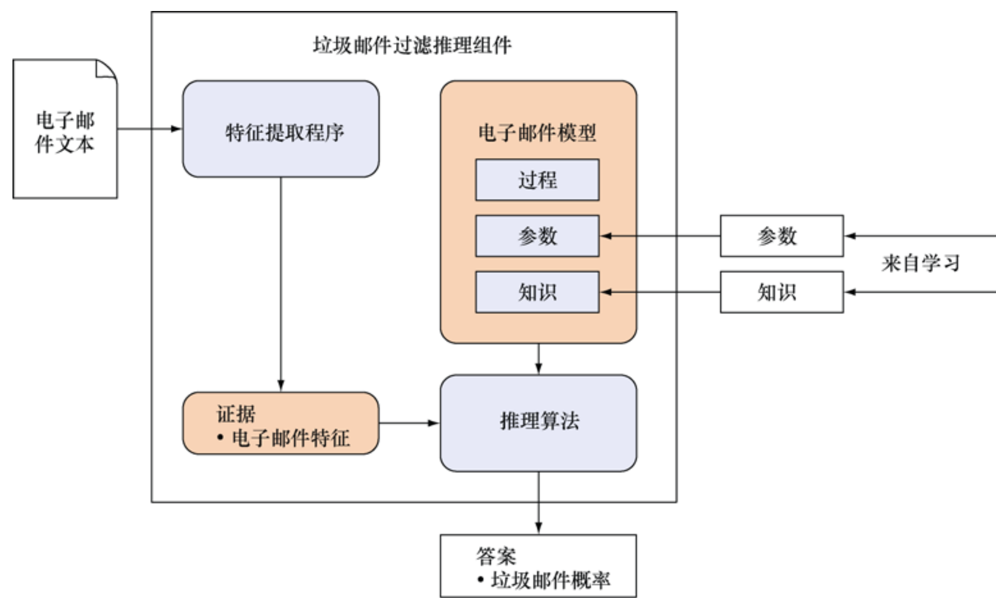


图 3-9　推理组件架构重现

程序清单 3-6　Email 类

```
class Email(file: File) {
  def getAllWords() = …

  val allWords: Set[String] = getAllWords()

  def observeEvidence(
    model: Model,
    label: Option[Boolean],
    learning: Boolean
  ) {
    label match {
      case Some(b) => model.isSpam.observe(b)
      case None => ()
    }

    for {
       (word, element) <- model.hasWordElements
    } {
      element.observe(allWords.contains(word))
    }

    val obsNumUnusualWords =
      allWords.filter((word: String) =>
        model.dictionary.isUnusual(word, learning)).size
```

用 I/O 操作（没有展示）获得电子邮件中的所有单词

观测证据的模型

根据电子邮件特征，在模型上观测证据的方法

可能存在也可能不存在的标签选项

表示当前是在学习还是推理的标志

如果标签存在，应用到模型的 isSpam 元素

对每个单词和模型中对应的元素，观察电子邮件的 allWords 是否包含该单词

统计文档中不寻常单词的数量

```
    val unusualWordFraction =
      obsNumUnusualWords * Model.binomialNumTrials / allWords.size
    model.numUnusualWords.observe(unusualWordFraction)
  }
}
```

对模型的 numUnusualWords 元素应用观测值

下面同样是上述代码的一些注释。

- 创建 Email 对象时，它在其文件参数上执行一些 I/O 操作，获取电子邮件中的所有单词。这些单词保存在 allWords 字段中。注意，allWords 是一个集合，与您的模型保持一致，在模型中您只关心某个单词是否出现在电子邮件中，而不是其出现次数。
- observeEvidence 的参数 label 类型为 Option[Boolean]，表示 label 是可选的。Scala 的 Option 数据结构可以取 Some(c)或者 None 值。在本例中，如果电子邮件有一个已知的分类，c 就是该分类。如果电子邮件分类未知，标签为 None。如果标签存在（标签选项值为 Some(c)），它将作为证据应用到 model.isSpam。
- 文档中不寻常单词的计数使用了标准的 Scala filter 方法。这里，filter 方法取得字符串集合 allWords，删除不满足其参数的所有字符串。filter 的参数是一个函数，如果单词是不寻常的则返回 true。所以，filter 方法删除所有不是不寻常单词的字符串。然后，取得结果集大小以得到不寻常单词的数量。注意，isUnusual 方法以学习标志作为第二个参数。
- 最后三行声明一个关于不寻常单词数量的观测值。记住，numUnusualWords 是由一个二项分布定义的，其中的试验次数由 Model.binomialNumTrials 给出。所以，您将 obsNumUnusualWords 按比例缩小为 Model.binomialNumTrials 的对应分数。

现在，您已经知道如何提取特征和应用证据，并且拥有了包含学习到的参数的模型，可以将电子邮件分类为正常邮件或者垃圾邮件。可以使用任何推理算法完成分类，您将使用一个精确算法——变量消除法（VE）。在单个电子邮件上，这个算法速度能够满足我们的要求，因为它是精确算法，可以得到最佳的答案。下面是代码。

程序清单 3-7　classify 方法

```
def classify(
  dictionary: Dictionary,
  parameters: LearnedParameters,
  fileName: String
) = {
```

classify 方法取得字典和参数形式的学习结果，以及包含待分类电子邮件的文件名称，返回电子邮件是垃圾邮件的概率

```
    val file = new File(fileName)
    val email = new Email(file)

    val model = new ReasoningModel(dictionary, parameters)

    email.observeEvidence(model, None, false)

    val algorithm = VariableElimination(model.isSpam)
    algorithm.start()

    val isSpamProbability = algorithm.probability(model.isSpam, true)
    println("Spam probability: " + isSpamProbability)

    algorithm.kill()

    isSpamProbability
}
```

从 fileName 指向的文件创建 Email 对象

用学习结果创建推理模型

观察模型上关于这个特定电子邮件的证据。因为您不是在学习，可选的标签为 None，学习标志为 false

创建和运行 VE 算法，以模型的 isSpam 元素为查询目标

获得该邮件是垃圾邮件的概率并打印对应的消息

杀死算法，进行清理。在获得需要查询的答案之前，一定不能杀死算法

返回该电子邮件是垃圾邮件的概率

最后，推理应用有一个 main 方法。这个方法在从命令行运行推理应用时被调用。

程序清单 3-8　Reasoning component main method

```
def main(args: Array[String]) = {
  val emailFileName = args(0)

  val learningFileName = args(1)

  val (dictionary, parameters) = loadResults(learningFileName)

  classify(dictionary, parameters, emailFileName)
}
```

从第 1 个命令行参数获得电子邮件文件名

从第 2 个命令行参数获得包含学习结果的文件名

用 loadResults 命令从学习文件中加载学习结果（未展示）

使用学习结果分类给定文件中的电子邮件

我没有展示 loadResults 方法的代码，因为它是常规的 I/O，十分冗长。如果对此感兴趣，请查看存储库上的代码。这些代码完成如下任务。

1. 读入模型的常规参数，如垃圾邮件概率和电子邮件有许多不寻常单词的概率。

2．读入所有单词和它们出现的次数。

3．读入特征单词和每个单词在垃圾邮件或者正常邮件中出现的概率。

该文件的格式是专为这个特殊应用设计的，Figaro 没有用于学习结果的通用格式。

3.6　创建学习组件

在定义模型、构建推理组件之后，您已经拥有了学习组件的大部分。我们再来看看图 3-10 中重现的学习组件架构。您已经看到了电子邮件模型。如前所述，学习组件中的电子邮件模型和用于推理的模型之间有一个差异——学习模型使用先验参数，该参数由用于模型各个参数的 Beta 元素组成。

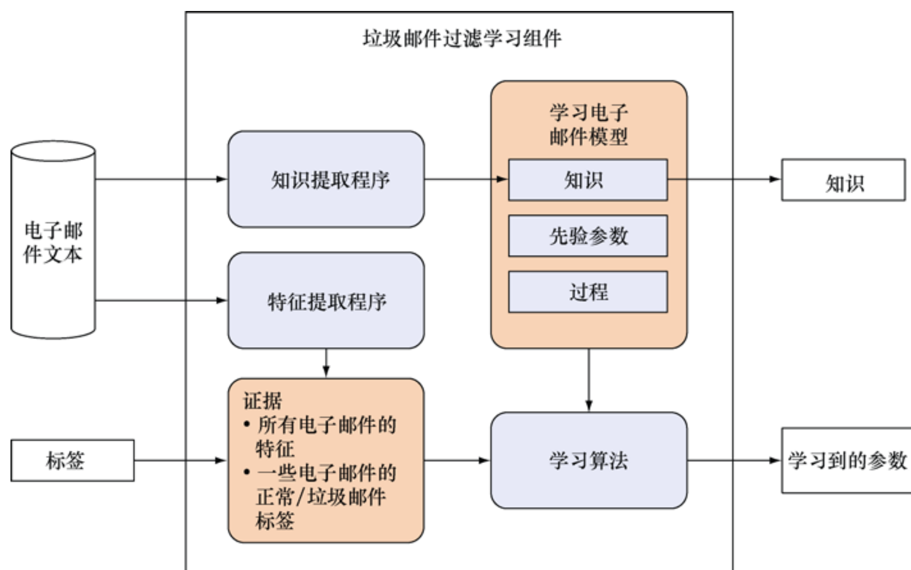


图 3-10　学习组件架构重现

您已经看到推理组件中的特征提取程序以及电子邮件和可选的正常/垃圾标签给定时证据的应用方式。在推理组件中，您进行单一电子邮件的推理。相反，在学习组件中，您有一个完整的电子邮件训练集。特征提取程序将提取所有电子邮件的特征，编辑每个电子邮件的证据。它还读取包含训练集中某些邮件标签的 Labels 文件，并在包含标签时应用对应的证据。

观察图 3-10 中的架构，可以看到现在有一个知识提取程序，其任务是从电子邮件文本中得到辅助知识。您已经看到这种知识由一个字典组成，并且看到 Dictionary 类有一个 addEmail 方法，所以垃圾邮件过滤器中的知识提取程序是很简单的。您定义一个名为 Dictionary 的 Scala 对象。在 Scala 中，对象就像只有一个实例的类。它可用于模拟 Java 静态方法。您将定义一个 fromEmails 方法，从 Emails 的 Traversable（Traversable 是许多集合类型（如列表和集合）的 Scala 基类）构造一个 Dictionary，可以使用 Dictionary.fromEmails(emails)调用该方法。注意，您已经有了 Dictionary 类，同样的名称可以同时用于类和对象。

程序清单 3-9　Dictionary 对象

```
object Dictionary {
  def fromEmails(emails: Traversable[Email]) = {
    val result = new Dictionary(0)
    for { email <- emails } { result.addEmail(email) }
    result
  }
}
```

创建 Dictionary 类的一个实例，其中电子邮件初始数量为 0。将每个电子邮件加入该字典并返回

最后来到学习组件的主要部分，这是应用学习算法学习模型参数的代码。您将使用的学习算法是**期望最大化（EM）**方法。EM 是一种“元算法”，因为它的内循环中使用常规的推理算法。您将在第 12 章中学习 EM 的各方面知识。概要地讲，EM 从对参数值的猜测开始，用这一猜测计算模型中元素的概率，然后使用这些概率改善猜测。这一过程可以重复任意多次，这个推理算法用于计算元素概率的步骤中。

Figaro 使用推理算法提供了 EM 的实现。在这个应用中，您将使用**置信传播**（belief propagation，BP）。BP 算法与 VE 类似，但是在大模型上工作速度更快。BP 在元素之间传播消息，并且进行一定数量的消息传播迭代。运行包含 BP 的 EM 时，需要选择 EM 和 BP 的迭代次数。对本应用，您将使用默认值 10，该值在本例中很合适。

应用学习算法的代码在下面的 learnMAP 方法中，以先验参数为参数，返回学习到的参数。**MAP** 是我们使用的学习方法——**最大后验方法（maximum a posteriori）**的缩写：根据数据，返回后验概率最大化的参数值。这些 MAP 值由 EM 算法生成。

程序清单 3-10　learnMAP 方法

```
def learnMAP(params: PriorParameters): LearnedParameters = {
  val algorithm =
    EMWithBP(params.fullParameterList:_*)
  algorithm.start()
```

实例化和运行使用 BP 计算概率的 EM 算法。使用默认 EM 和 BP 迭代次数

```
    val spamProbability = params.spamProbability.MAPValue
    val hasUnusualWordsGivenSpamProbability =
      params.hasManyUnusualWordsGivenSpamProbability.MAPValue
    val hasUnusualWordsGivenNormalProbability =
      params.hasManyUnusualWordsGivenNormalProbability.MAPValue
    val unusualWordGivenHasUnusualProbability =
      params.unusualWordGivenManyProbability.MAPValue
    val unusualWordGivenNotHasUnusualProbability =
      params.unusualWordGivenFewProbability.MAPValue

    val wordGivenSpamProbabilities =
      for { (word, param) <- params.wordGivenSpamProbabilities }
      yield (word, param.MAPValue)
    val wordGivenNormalProbabilities =
      for { (word, param) <- params.wordGivenNormalProbabilities }
       yield (word, param.MAPValue)

    algorithm.kill()

    new LearnedParameters(
      spamProbability,
      hasUnusualWordsGivenSpamProbability,
      hasUnusualWordsGivenNormalProbability,
      unusualWordGivenHasUnusualProbability,
      unusualWordGivenNotHasUnusualProbability,
      wordGivenSpamProbabilities.toMap,
      wordGivenNormalProbabilities.toMap
    )
  }
```

计算每个参数元素最可能的 Double 值

对每个特征单词,计算最有可能的 Double 参数值,该参数表示单词出现在垃圾邮件和正常邮件中的概率。将这些值放到一个（单词,值）配对列表中

杀死算法进行清理。注意，在获得 MAP 值之前不要杀死该算法，因为它们在算法被杀死之后将失效

返回 Learned-Parameters 的一个实例，包含所有学习到的参数值

最后，学习组件有一个 main 方法，在从命令行运行学习程序时调用，这个方法取得 3 个参数：包含训练集中所有电子邮件的目录名称、包含一些训练样本标签的文件名和保存学习结果的文件名。它首先从输入文件中读入信息并提取字典。接下来，为每个电子邮件创建模型。最后，它运行学习算法并输出结果。这些结果可供推理组件使用任意次。

值得注意的是，所有电子邮件共享相同的参数。这使从所有电子邮件学习相同的参数成为可能。但是，每个电子邮件在模型中有不同组元素。例如，每个电子邮件的 isSpam 元素不同。如果它们不是各不相同，所有电子邮件对于该元素就有相同值，也就是说，都是垃圾邮件，或者都是正常邮件。另一方面，每个不同电子邮件的模型在结果和参数上都是完全相同的。

下面是 main 方法的代码。

程序清单 3-11　学习组件的 main 方法

```
def main(args: Array[String]) {
  val trainingDirectoryName = args(0)
  val labelFileName = args(1)
  val learningFileName = args(2)

  val emails = readEmails(trainingDirectoryName)

  val labels = readLabels(labelFileName)

  val dictionary = Dictionary.fromEmails(emails.values)

  val params = new PriorParameters(dictionary)
  val models =
    for { (fileName, email) <- emails }
    yield {
      val model = new LearningModel(dictionary, params)
      email.observeEvidence(model, labels.get(fileName), true)
      model
    }

  val learnedParameters = learnMAP(params)

  saveResults(learningFileName, dictionary, learnedParameters)
 }
}
```

从训练目录读取所有电子邮件。readEmails（没有展示）返回文件名到 Email 实例的一个映射

从标签文件读取标签。readLabels（没有展示）返回从文件名到标签的映射。没有标签的任何文件不会出现在这个映射中

从 emails 映射值构造字典

构建模型，观察证据。注意，所有模型共享相同的先验参数，但是有 LearningModel 的不同实例。还要注意 observeEvidence 的可选标签参数，您使用的是 labels.get（filename）。如果没有标签，该值为 None，如果标签是特定分类，则为 Some（class）

运行学习算法

用 I/O 命令将学习结果保存到一个文件，供推理组件使用（没有展示）

您已经看到了一个完整的应用程序，从架构开始，然后经历模型设计、推理组件实现和学习组件的实现。现在，您应该基本掌握了 Figaro 的特性。您已经了解如何构建一个应用，有了一定的能力。

本书的下一部分更深入地介绍建模方法。首先，您将更深入地理解概率模型和推理。然后，将研究各种使用 Figaro 创建丰富、实用概率模型的创造性方法。

3.7　小结

- 概率编程往往用在从训练数据学习模型，然后对特定实例应用该模型任意次的应用程序中。
- 许多概率编程应用都有类似的架构，包含推理组件和学习组件。

- 推理和学习组件都使用一个模型，该模型包含生成过程的规范、过程的参数和辅助知识。
- 学习组件使用先验参数并学习得到这些参数的值，这些值可以供推理组件使用。

3.8 练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

本章的练习是高级思维练习，要求您考虑如何按照本章中例子的思路，设计一个概率推理和学习应用以解决现实问题。当您做这些练习时，考虑总体架构和设计，无需操心模型的细节。

1．您打算构建一个智能图像搜索引擎，搜索包含感兴趣的物体的图像，这些图像甚至没有明确地标示出那些物体。您的搜索引擎将包含一个物体识别器，可以标记出图像中包含的物体。您决定以概率推理和学习应用的形式开发这个物体识别器。

a）粗略描述搜索引擎的架构和物体识别器所起的作用。

b）粗略描述推理组件的架构。识别器的输入和输出是什么? 您不希望识别器处理原始像素，而是首先提取图像的高级特征，如色彩直方图和边缘检测。如何将这些特征融入到架构中?

c）粗略描述学习组件的架构。输入和输出是什么？学习结果如何传达给推理组件？

2．您打算设计一个安全防护应用组件，其任务是检测百货公司中的异常活动。在您的方法中，异常活动定义为低概率活动。您将学习一个网络活动的概率模型。每当在网络中发现活动，就计算这种活动的概率。如果概率很低，就发出异常的信号。

a）异常检测程序的输入和输出是什么？如何融入到整个安全防护应用中？

b）在这个设计中，您没有表示活动分类的变量，而是在描述活动的变量之上创建一个概率模型。有哪些此类变量？在变量之间存在哪些类别的关系和依赖性？

c）您的异常检测程序训练集包含哪些内容？可以从这些数据学习到哪类知识？

3．您打算设计一个智能扑克玩家程序，目标有二：在您第一次与对手打扑克时能有很好的表现；随着时间的推移，您的表现会变得更好。

a）扑克中有两种随机性的来源：发牌和玩家的出牌。描述与扑克玩家程序相关的一些变量，并用随机性来源标记。从学习应用的角度看，为什么您认为一个变量是发牌和玩家出牌的结果有所不同?

b）在扑克玩家程序中，您希望实现两种学习。首先，您希望学习人们的一般玩法。这将帮助您在第一次遇到对手时有出色的表现。其次，当您重复地和一位特定的玩家打扑克时，您希望学到该玩家的倾向。描述能够执行这两类学习的学习系统架构。在不同组件之间需要传达哪些信息?

第 2 部分

编写概率程序

第 2 部分介绍编写概率程序以表示您所感兴趣的情况的相关知识。这一部分的目标不仅是为您提供编写程序的工具，还要确保您理解这些程序的含义和选择使用某种编程技术的原因。第 4 章简单介绍概率建模和概率编程的基本思路；尽管这一章中没有多少程序，但是它能够帮助您打下良好的基础，理解基本原理，从而更加接近编写概率程序的任务。第 5 章介绍两种主要的建模范式，它们是概率编程的核心：贝叶斯网络和马尔科夫网络。第 6 章～第 8 章在第 4 章和第 5 章的基础上介绍更高级的建模技术，包括使用集合、面向对象编程和动态系统建模。

第 4 章　概率模型和概率程序

本章介绍如下内容：

- 概率模型定义
- 如何使用概率模型回答查询
- 概率模型的成分，包括变量、依赖性、函数形式和数值参数
- 概率程序如何表示概率模型的各个成分

本书的第 1 部分介绍了概率编程。您已经学到，概率推理系统使用概率模型，根据证据回答查询，概率编程使用程序表示概率模型。本书的这个部分进一步深入概率模型的表现。您将学习到编写概率程序所用的各种编程技巧。

但是，您首先要拓展对概率模型以及构造和使用它们以回答查询的方法的基本理解。本章就提供这些理解。在第 1 部分中，您已经直观地使用了其中一些思路，现在是时候了解这些基础知识了。

本章更深入地详述第 1 章的主题，所以如果您认真阅读了第 1 章，对接下来的学习就很有益处。本章还描述概率程序（特别是 Figaro 程序）如何定义概率模型，所以第 2 章中的 Figaro 基础知识也很有帮助。

4.1　概率模型定义

概率模型是编码对某个不确定情况一般知识的一种方式。例如，想象您是一位艺术专家，试图决定一幅画是伦勃朗的真迹还是赝品。您有如下的信息。

1．伦勃朗喜欢在画作中使用深颜色。

2．他常常创作人物画。

3．新发现的大师真迹很稀有，但是赝品多如牛毛。

4．这幅画使用了大片的嫩黄色。

5．这幅画是一位水手的肖像。

6．这幅画在 2003 年的一次拍卖中售出。

第 1～3 项是**一般**知识。第 4～6 项是**现时**知识。一般知识以趋势的形式描述：“喜欢使用”“往往”“很少”“多如牛毛”。现时知识很具体：“这幅画使用了”“这幅画是…的肖像”“这幅画在…售出”。在概率建模中，您在概率模型中编码一般知识，并将其应用到现时知识以推理所处理的具体情况。现时知识也称作**证据**，而您试图找出的事实（这幅画是不是伦勃朗的真迹）是**查询**。所以，在此您试图将关于绘画和伦勃朗的一般知识（包含在断言 1～3 中）应用到断言 4～6 中编码的证据，以回答这幅画是否是真迹的查询。

4.1.1　将一般知识表达为可能世界上的某种概率分布

图 4-1 展示了一般领域知识的例子。一般知识说明两个事实：(1) 可能的情况，(2) 可能性较大的情况。首先，我们讨论一下可能的情况：当您创建概率模型时，将设想许多可能的状态——每一个状态称作一个**可能世界**。例如，一幅画是赝品就是一个可能世界；另一个可能世界则是，它是伦勃朗的真迹。每个可能世界描述看到任何证据之前认为可能的一种情况。 例如，伦勃朗的风景画在您看到任何证据之前是一个可能世界。后来，如果您看到的证据是“这是一幅人物画”，就会使用该证据排除上述可能世界。可能世界是一般可能发生的情况：伦勃朗可能有风景画。证据描述现时知道的情况：这是一幅人物画。

例如，在这个情况中，您可以构想如下可能世界。

- w_1——这幅画是赝品。
- w_2——这幅画是用深颜色描绘人物的伦勃朗真迹。
- w_3——这幅画是用鲜艳颜色描绘人物的伦勃朗真迹。
- w_4——这幅画是伦勃朗风景画的真迹。

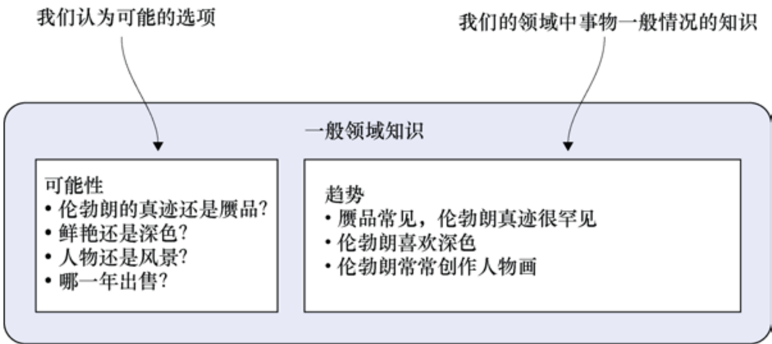


图 4-1　一般领域知识包括可能的情况和可能性较大的情况

注意，可能世界是您在看到任何证据之前认为可能的情况。重要的是要理解，证据不是概率模型的一部分。模型本身描述一般知识。概率推理使用编码一般知识的模型，将其应用到关于某个特定情况的证据，以便回答关于那种情况的查询。因为证据不是模型的一部分，模型由可能世界组成，所以不能用证据决定可能世界。因此，您会得到 w_2 和 w_4 这样的可能世界，它们与陈述 4 和 5 的证据冲突。

现在，我们来谈谈一般知识中包含的第二种信息：可能性较大的情况。这通过为每个可能世界指定一个数值进行编码，这个数值称为**概率**。表 4-1 展示了可能世界 w_1～w_4 的概率，所有可能世界的概率加起来必须为 1。可能世界的概率分配称为**概率分布**。这个术语的含义是为可能世界分配概率，不管这种分配基于何种知识。

表 4-1　通过为可能世界分配加总为 1 的概率，定义概率分布

可 能 世 界	描　　述	概　　率
w_1	赝品	0.9
w_2	伦勃朗、人物、深色	0.09
w_3	伦勃朗、人物、鲜艳	0.009
w_4	伦勃朗、风景	0.001

图 4-2 总结了取得一般知识（包括所有设想的可能性和可能性中的趋势），并将其表现为可能世界上概率分布的过程。这些可能性决定可能世界的定义。然后，趋势（可能性较大的情况）决定了每个可能世界的概率。

您已经理解了可能世界和概率分布，是时候定义概率模型了。概率模型是可能世界上概率分布的正式表现形式。

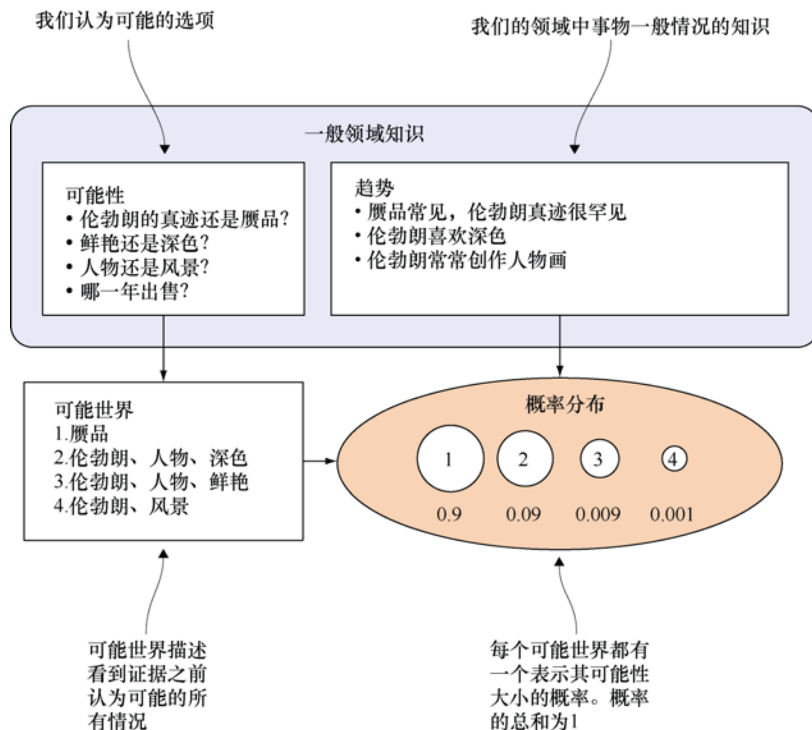


图 4-2　领域知识以可能世界上的概率分布的形式表现。在图中，概率由可能世界的大小表示

上述定义并不是说概率模型就是一个概率分布，而是说明它是这样一个分布的**表现形式**。概率分布本身是一个数学概念，明确地向每个可能世界指定一个数字。可能世界的数量可能很大，分布必须为每个可能世界指定一个概率。这种表现形式可以写下来并加以处理。最简单的概率模型就是一个明确列出可能世界概率的表格，但是正如您在本书中看到的，即使在可能世界的数量很大时，模型也可以相当简洁。上述的定义确实说到，概率模型必须是正式的表现形式。即使没有明确地创建概率，也应该用清晰的规则规定每个可能世界概率的定义形式。

虽然我已经定义了一个概率模型，但是还没有描述创建该模型的方法——如何指定可能世界和概率。概率建模的艺术和科学都与以精确、简洁的方式表现概率分布有关。您将在本书中学到这方面的所有方法。

您已经了解了概率分布对于概率建模的重要性，下面我们更仔细地研究这一概念。

4.1.2　进一步探索概率分布

除了告诉您各个可能世界的概率之外，概率分布还可以告诉您不同事实的概率。任何指定的事实都包含在某些可能世界中，而不包含在其他可能世界。为了获得该事实的

概率，您可以将包含该事实的可能世界的概率累加起来。

例如，考虑“这幅画是伦勃朗的真迹”这一事实。这一事实包含在 w_2、w_3 和 w_4 中，但不包含在 w_1 中，它的概率等于包含该事实的可能世界概率之和，也就是 w_2、w_3 和 w_4 的概率总和：0.09 + 0.009 + 0.001 = 0.1。因此，您可以说这幅画是伦勃朗真迹的概率为 0.1 或 10%。结论的标准标记方式为 P(Rembrandt) = 0.1，如图 4-3 所示。

在您看到任何证据之前，这种概率分布称为**先验概率分布**。顾名思义，它先于任何现时因素的考虑（在看见证据之前）。与此相反，**后验概率分布**是考虑证据之后得到的分布。同样，上一段中计算的 P(Rembrandt)称作**该事实的先验概率**，而在观察证据之后将得到后验概率。您将在下一小节中看到如何将证据考虑在内，获得后验概率。

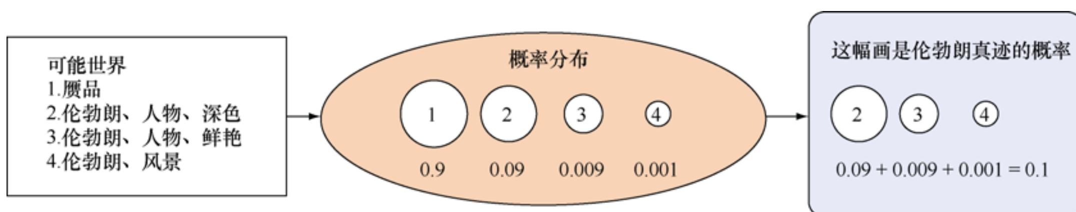


图 4-3 某个事实的概率是与该事实一致的可能世界的概率总和

列举可能世界并指定概率的方法很多。出现在表 4-1 并在表 4-2 中再次出现的概率分布恰好符合本节开始时项目 1、2 和 3 描述的一般知识。项目 3 说明，赝品比伦勃朗的真迹更常见，确实，可能世界 w_1 中赝品为“真”，其概率为 0.9，而在其他 3 个可能世界中“伦勃朗真迹”为“真”，其总概率仅为 0.1。项目 1 说明伦勃朗喜欢使用深色。如果观察表格，就会看到 w_2（伦勃朗、人物、深色）的概率 10 倍于 w_3（伦勃朗、人物、鲜艳）。尽管剩下的可能世界 w_4 不说明任何关于颜色的情况，但是它在先验分布中概率极低，仅为 0.001。最后，项目 2 说明伦勃朗常常创作人物画。w_2 和 w_3 与“这幅画是人物画”的事实一致，所以根据我们的模型，这幅画是伦勃朗真迹且是人物画的概率等于 w_2 和 w_3 概率的总和——0.099。与此同时，这幅画是伦勃朗所作的风景画的概率仅为 0.001，只有人物画的 1/99。

表 4-2 重温我们的概率模型。赝品的可能性远大于伦勃朗真迹的可能性，所以 w_1 的概率超过其他 3 个可能世界的总和。同样，如果画家是伦勃朗，所创作的更可能是人物画而非风景画，深色的人物画比鲜艳的更有可能

可 能 世 界	描 述	概 率
w_1	赝品	0.9
w_2	伦勃朗、人物、深色	0.09
w_3	伦勃朗、人物、鲜艳	0.009
w_4	伦勃朗、风景	0.001

在这个例子中，您不用任何真正的方法就可以组成一个看似合理的模型，但是在更复杂的情况下，这就更难了。在 4.3 节，您将开始研究更结构化的概率分布表示方法。

但是，在进入模型表示的细节之前，您必须首先理解说明如何使用模型推理特定情况的基本术语。

4.2　使用概率模型回答查询

您已经以可能世界上概率分布的形式编码了一般知识。如何将模型应用到证据（现时知识）以回答查询？例如，您知道这幅画上有一大片嫩黄色，是水手的肖像，在 2003 年的一次拍卖中售出。这些都是您的证据。您希望知道这幅画是不是伦勃朗的真迹——那是您的查询。如何使用模型，按照证据推理出查询的答案？表 4-3 总结了您的知识状态。

表 4-3　在这个情况中组成知识的模型和证据。每部分证据都用其与模型的相关性注释

模　　型		证　　据
赝品	0.9	大片的嫩黄色==>鲜艳
伦勃朗、人物画、深色	0.09	水手的肖像==>人物画
伦勃朗、人物画、鲜艳	0.009	2003 年出售（无关）
伦勃朗、风景	0.001	

4.2.1　根据证据调节以产生后验概率分布

使用概率分布以考虑证据的过程称作**根据证据调节**。调节的结果也是一个概率分布，称作**后验概率分布**。

调节过程很简单，由两个步骤组成。

1. 消除所有与证据不一致的可能世界。

观察您的可能世界，可以看到其中两个与证据不一致：w_2（使用深色）和 w_4（风景画）。所以您“删去”这些可能世界，将它们的概率指定为 0。这幅画在 2003 年的拍卖上售出的事实不能排除任何可能世界，这部分证据与模型无关。

2. 向上调整剩余可能世界的概率使其总和为 1。

这称作概率的**规格化**。在规格化时，剩余的可能世界必须吸收被排除可能世界的概率。每个可能世界吸收的概率数量与其先验概率成正比。例如，因为 w_1 的先验概率是 w_3 先验概率的 100 倍，被排除的可能世界吸收到 w_1 的概率也是吸收到 w_3 的 100 倍。这确保了吸收之后，w_1 的概率仍然是 w_3 的 100 倍。

正确规格化概率的数学方法很简单。首先，累加与证据一致、未被排除的可能世界

概率。这个总和称为**规格化因子**。在我们的例子中，规格化因子是 0.9+0.009=0.909。然后，将每个一致可能世界的概率除以规格化因子。这将保证概率总和为 1 且维持比例。这一过程的结果是表 4-4 中的概率分布。

表 4-4　根据证据调节之后得到的后验概率分布。首先，将不一致的可能世界概率减为 0。然后，将剩余的每个概率除以一致可能世界的概率总和，进行规格化。这确保概率的总和为 1

可 能 世 界	描　　述	概　　率
w_1	赝品	0.9 / 0.909 = 0.9901
w_2	伦勃朗、人物画、深色	0
w_3	伦勃朗、人物画、鲜艳	0.009 / 0.909 = 0.0099
w_4	伦勃朗、风景	0

证据的调节由排除与证据不一致的可能世界和规格化其余概率组成。根据证据调节之后得到的概率分布称作后验概率分布，因为它在看到证据之后出现。图 4-4 描述了从先验概率分布开始，根据证据调节并获得后验概率分布的过程。

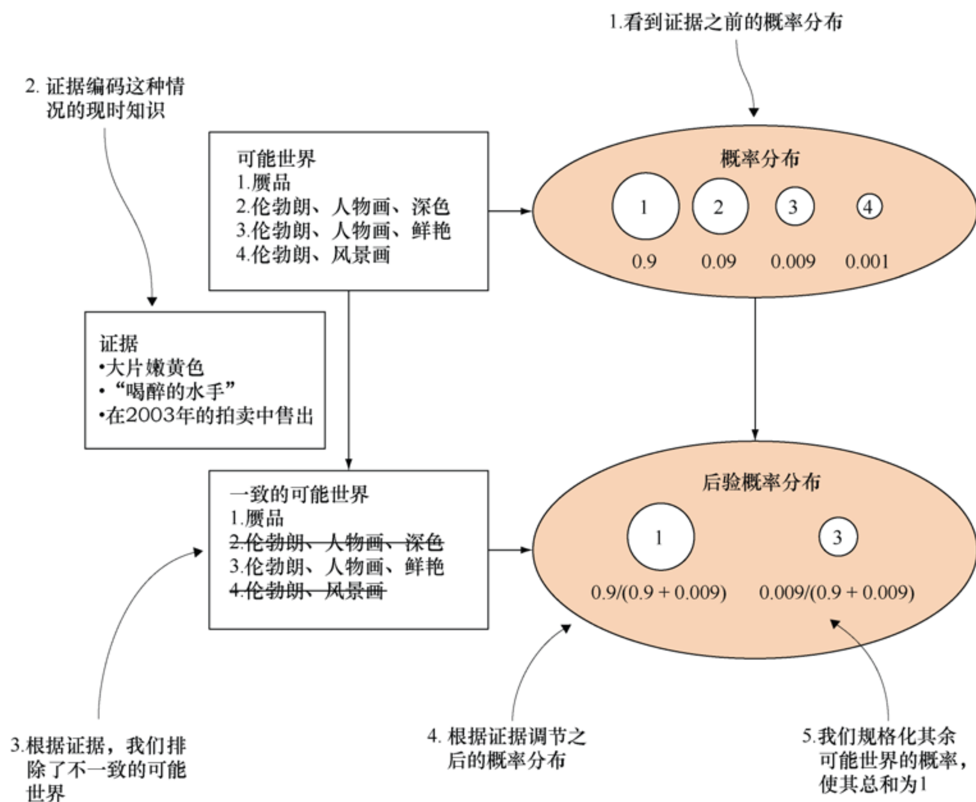


图 4-4　根据证据调节。根据证据，排除不一致的可能世界并规格化剩余可能世界的概率，使之总和为 1

4.2.2　回答查询

现在，您拥有了后验概率分布，如何回答查询？在我们的例子中，查询是“画家是不是伦勃朗”。在前一小节中我说过，概率分布不仅指定单独可能世界的概率，还指出任何包含在某些可能世界而不包含在其他可能世界的事实的概率。例如，w_3 是与画家是伦勃朗一致且后验概率大于 0 的可能世界。所以画家是伦勃朗的后验概率是 0.0099。

您为根据证据得出的查询答案使用如下的标记法：P（伦勃朗 | 嫩黄色，水手肖像，售于 2003 年）= 0.0099。在这个标记法中，管道符号（|）将查询与证据分隔开。管道符号左侧是您想知道概率的事物。右侧是您得到的证据。这种标记法也称作**条件概率**，因为它表示根据某些证据调节过的查询概率。图 4-5 展示了这样构建条件概率陈述的方法。

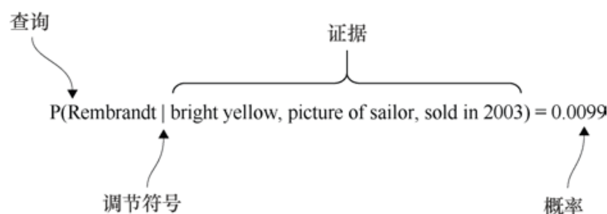


图 4-5　条件概率陈述的结构

警告：如果您只观察标记法，似乎可以对认为不可能的证据（概率为 0）进行调节。例如，如果您尝试观察“这幅画是伦勃朗的静物画”的证据，将发生什么情况？假定您希望查询画作的鲜艳度。使用条件概率标记法，可以尝试计算 P（鲜艳|静物，伦勃朗）。但是“这幅画是伦勃朗的静物画”的证据概率为 0，因为它与所有可能世界不一致。当您根据证据调节时，将排除所有可能世界。因此，您无法产生后验概率分布，也不能对“这幅画是鲜艳的”的概率做出任何结论。根据不可能的证据调节耗费了许多精力，结论却是应该避免这么做。如果在不可能的证据上进行调节，结果将不可预测，在不同的概率编程系统上各不相同。

图 4-6 组合前两节的所有思路，组成概率建模的全景。您使用一般领域知识编码可能世界上的概率分布。然后，您的现时知识以证据的形式出现，用于产生后验概率分布。最后，使用后验概率分布，按照证据回答关于感兴趣事实的查询。

关键定义

可能世界——您认为可能的所有状态。
概率分布——为每个可能世界分配 0～1 的概率，所有概率总和为 1。
概率模型——可能世界上概率分布的正规表现形式。
证据——关于特定情况的知识。

先验概率分布——看到任何证据之前的概率分布。

根据证据调节——将证据应用到某个概率分布的过程。

后验概率分布——看到证据之后的概率分布，调节的结果。

规格化——按比例调整一组数字使其总和为 1 的过程。

现在，您已经知道概率模型的定义以及回答查询的基本原理。但是，如何将这些理论应用到实践中呢？可能世界的数量可能很大；检查每个可能世界，查看其是否与证据一致是不实际的。概率推理算法的目标是高效地回答查询。

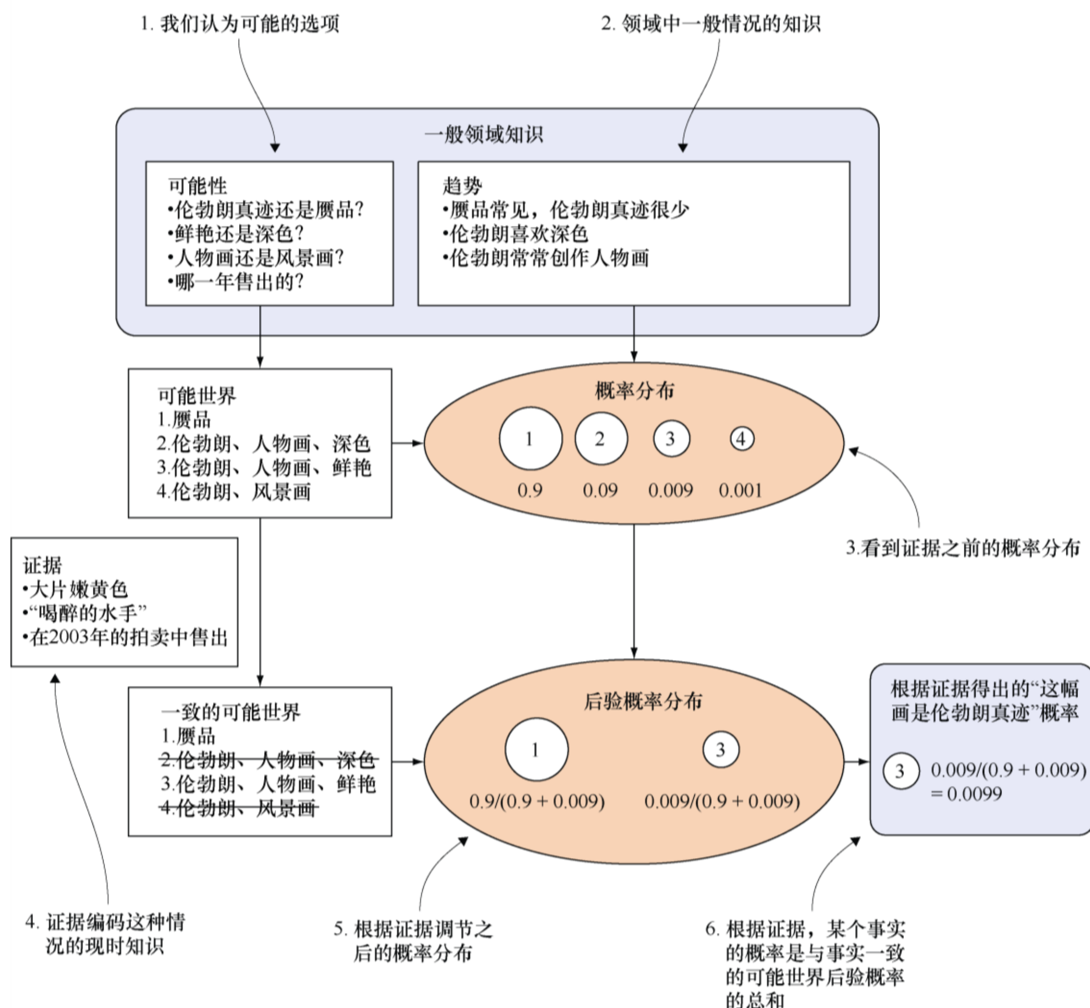


图 4-6　概率模型使用方法全貌

4.2.3 使用概率推理

概率推理的主要目标是根据证据，计算感兴趣的变量上的后验概率分布。在我们的例子中，排除与证据不符的可能世界、规格化其余可能世界的概率很简单。但是在实践中，对于真正的问题，可能世界的数量巨大；模型中的变量呈指数式上升。因此，您甚至无法写下所有可能世界，更别说计算它们的概率了。您需要一个概率推理算法，可以高效地计算后验概率。Figaro 等概率编程系统提供了此类算法。

这些算法基于概率推理的三大法则：链式法则、全概率公式和贝叶斯法则。您将在第 3 部分中学习这些法则的细节。现在，您只需知道概率编程推理算法是实施这些法则和其他派生法则的机械方法。这些算法设计用于和定义它们的概率编程语言结构一起工作，所以可用于以这些语言编写的复杂程序中。这是概率编程的好处之一：如果可以用编程语言编写程序，就可以对该程序应用推理算法。

在实践中，尽管原理上算法有效，但是概率推理可能很困难。对于常规程序来说，很容易写出具有指数级复杂度、在您所关心的问题上花费很长时间的程序。类似地，也很可能写出推理算法为产生精确答案而长时间运行的概率程序。对于许多典型使用来说，可以编写自己的程序，推理将按照您需要的方式进行。但是，在复杂的大问题上，即使最好、最高效的算法也可能需要长时间才能得出答案。

概率推理算法可以是精确的，也可以是近似的。**精确**意味着它们计算的后验概率从数学上遵循推理的 3 大法则。精确推理代价可能很高，所以需要近似算法。**近似**算法的结果通常接近于正确答案，但不总能提供保证。

概率编程系统往往为给定的问题提供可选择的算法，包括精确算法和近似算法。有些算法可以不同的方式配置。此外，表达模型的方式可能对推理的难度产生显著影响。所以，如果您打算使用概率编程语言构建大型的复杂模型，值得花时间学习推理的工作原理，获得帮助您选择更有效解决自身问题最佳推理算法的技术。本书的第 3 部分将帮助您获得这方面的知识。

4.3 概率模型的组成部分

概率模型是概率分布的正式表现形式，这类表现形式可能多种多样。明确概率的表格是一种表现形式，但是它仅对最简单的问题适用。本节介绍使用 4 种成分构建概率模型的通用、实际方法。这不是构建模型的唯一方法，但是得到了广泛的应用，也是概率编程的基础。

您已经在第 3 章的垃圾过滤器模型中看到了这 4 种成分，但是在此将了解它们的细节。模型有 4 种成分。

- 涉及的**变量**，如一幅画是伦勃朗真迹还是赝品。
- 变量之间的**依赖性**。例如，画作的鲜艳度取决于它是不是伦勃朗的画。
- 这些依赖性采用的**函数形式**。例如，“这幅画是伦勃朗的真迹”的模型可以一定权重的“掷硬币”结果表示。
- 函数形式的**数值参数**，如用于确定画作是不是伦勃朗真迹的硬币权重。

在下面的几个小节中，您将看到如何指定以上成分。

4.3.1　变量

第一步是决定模型中所要包含的变量。正如编程语言中一样，**变量**可以取不同的值。可能世界为每个变量指定一个特定值。在任何给定的可能世界中，变量有一个特定值，但是在其他可能世界中，它可能取不同的值。虽然指定变量只是构建概率模型的第一步，但是和编程中一样需要很多创意。您所做出的决策对模型的效率至关重要。

在 4.1 节的例子中，我们指定的可能世界隐含了如下变量。

- Rembrandt（伦勃朗）：这幅画是伦勃朗的真迹还是赝品？
- Brightness（鲜艳度）：它使用鲜艳的颜色还是深色？
- Subject（主题）：是人物画还是风景画？

注意，我们没有包含表示其是否在拍卖上出售的变量。尽管该变量是现时知识的一部分，但是我们认为它不相关。

每个变量都有**类型**，定义了可能取值的种类。**连续**变量取实数值（如双精度变量）——例如，高度可以厘米计量。您还可以使用枚举、整数或者结构化变量（如列表）。不连续的变量称作**离散**变量。到目前为止看到的所有变量都是离散变量，更准确地说，是枚举变量。枚举类型变量往往称作**分类**变量。变量还有一个**值域**，即该变量可能取的一组值。例如，某个变量可能是整数变量，但是您认为它只可能取 1 和 10 之间的值，所以它的值域就是整数 1～10。表 4-5 列出了在伦勃朗示例中可能使用的变量、类型和可能取值。

表 4-5　变量、类型和可能取值示例

变　　量	类　　型	示　例　值
Rembrandt	布尔型	true, false
Size	枚举	small、medium、large
Height	实数	25.3, 14.9, 68.24
Last year sold	整数	1937, 2003
All years sold	整数列表	List(1937), List(1969, 2003)

概率模型中的变量和概率程序中的变量之间存在明显的关系。在 Figaro 中，程序中

的元素对应于模型中的变量，其值类型是模型中对应类型的计算机表现形式。例如，您可以这样定义：

```
val rembrandt: Element[Boolean] = // definition goes here
val size: Element[Symbol] = // definition goes here
val height: Element[Double] = // definition goes here
val lastYearSold: Element[Int] = // definition goes here
val allYearsSold: Element[List[Integer]] = // definition goes here
```

给定一组变量，构造可能世界的最简方式是创建一个表格，每列代表一个变量，每行代表变量的一个可能值。例如，表 4-6 展示了伦勃朗、鲜艳度和主题变量的 8 个可能世界。

表 4-6 伦勃朗、鲜艳度和主题变量的可能世界。每个可能世界代表这些变量值的一个组合

	伦 勃 朗	鲜 艳 度	主 题
w_1	假	深色	人物画
w_2	假	深色	风景画
w_3	假	鲜艳	人物画
w_4	假	鲜艳	风景画
w_5	真	深色	人物画
w_6	真	深色	风景画
w_7	真	鲜艳	人物画
w_8	真	鲜艳	风景画

4.3.2 依赖性

我已经声明，依赖性可用于为概率分布提供结构。**依赖性**描述了变量相互关联的方式。在所有可能世界上的大量概率分布直观地分解为描述其关系的局部模型组成部分。

回到我们的例子，现在您有 3 个变量：Rembrandt（伦勃朗）、 Brightness（亮度）和 Subject（主题）。这些变量如何关联？在概率模型中，变量之间的关系由依赖性描述，依赖性描述了一个变量值依赖于其他变量值的方式。**独立关系**和依赖性同样重要，前者描述了一个变量的值完全不依赖另一个变量值的情况。

理解独立关系

如果知道一个变量的有关情况，不能得出另一个变量的任何有关情况，那么这两个变量是独立的。例如，想象一下您有另一个变量 Size（可能值为大、中、小），表示画作的大小。“Size 和 Brightness 是相互独立的”有何含义？这一说法的含义很明确。

考虑 Brightness 的任何特定值——比如 Dark（深色）。您知道它的先验概率 P(Brightness = Dark)。现在考虑 Size 的可能值——比如 Large（大）。您打算提出一个问

题：知道 Size = Large 是否能够提供与 Brightness = Dark 相关的新信息。对于这个问题，证据是 Size = Large，查询是 Brightness = Dark，因为那是您想知道概率的事实。如果您根据证据 Size = Large 调节模型，就可以得到 Dark 的后验概率 P(Brightness = Dark | Size = Large)。现在，如果 Brightness 和 Size 是独立的，后验概率必然和先验概率相同。告诉您画作的尺寸为“大”丝毫不能改变您对画作是深色的看法。如果 Brightness 和 Size 是独立的，不管选择 Brightness 和 Size 的哪一个值，这一点都保持不变。下面的等式必须成立。理解这些等式的方法描述参见图 4-7。

P(Brightness = Dark | Size = Large) = P(Brightness = Dark)

P(Brightness = Dark | Size = Medium) = P(Brightness = Dark)

P(Brightness = Dark | Size = Small) = P(Brightness = Dark)

P(Brightness = Bright | Size = Large) = P(Brightness = Bright)

P(Brightness = Bright | Size = Medium) = P(Brightness = Bright)

P(Brightness = Bright | Size = Small) = P(Brightness = Bright)

独立性是一个对称的属性。如果您知道，告诉您画作的大小不能提供关于其鲜艳度的任何新信息，反方向上也是如此：告诉您画作的鲜艳度，也不能给画作的尺寸提供任何新信息。数学标记如下：

P(Brightness = Dark | Size = Large) = P(Brightness = Dark)

隐含：

P(Size = Large | Brightness = Dark) = P(Size = Large)

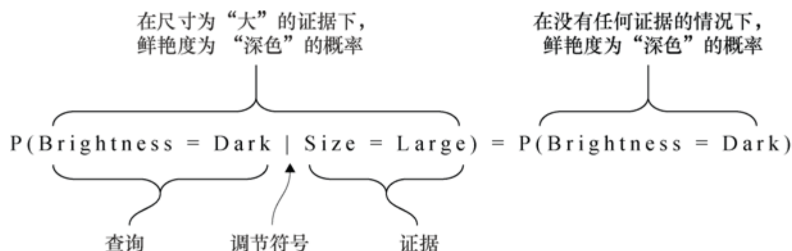


图 4-7　描述独立关系的等式分解

独立性等式如图 4-8 所示。您首先从 Size 和 Brightness 上的先验概率分布入手。Brightness 为 Dark 的概率是 Brightness = Dark 的可能世界概率的总和——0.8。然后，观察证据 Size = Small 并排除不一致的世界。通过规格化与证据一致的可能世界概率，获得 Brightness = Dark 的后验概率。后验概率为 0.8，与观察证据之前相同。如果观察先验分布就可以看出，对于任何 Size 值，Brightness 为 Dark 的可能世界概率与 Brightness 为 Bright 的可能世界概率之间的比率相同，均为 4∶1。因为比率相同，观察到某个 Size 的特定值不会改变观察到该证据之前的 Dark 与 Bright 比率。

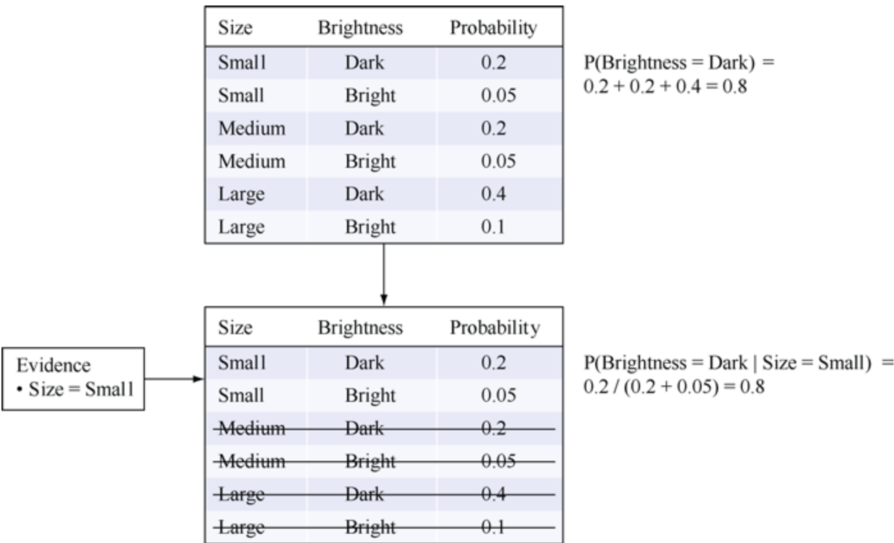


图 4-8 独立性：Brightness 为 Dark 的概率在观察到 Size 为 Small 之后没有变化

理解条件独立性

下一步是考虑两个变量在已知条件下的关系。某些情况下，两个变量在没有附加信息时可能不是相互独立的，但是在附加信息出现时可能相互独立。例如，假定风景画更有可能采用大尺寸和鲜艳。那么，Size 和 Brightness 就不是独立的，因为告诉您画作尺寸较大，就会增加画作是风景画的置信度，这也就使其更可能是鲜艳的。另一方面，如果您已经知道画作是一幅风景画，告诉您画作的尺寸较大不能提供它是什么类型的新信息，所以不会改变它颜色较亮的观点。在这种情况下，您可以说在 Subject（主题）=Landscape（风景画）的情况下，Brightness 和 Size **有条件独立**。上述情况可以用如下等式表示：

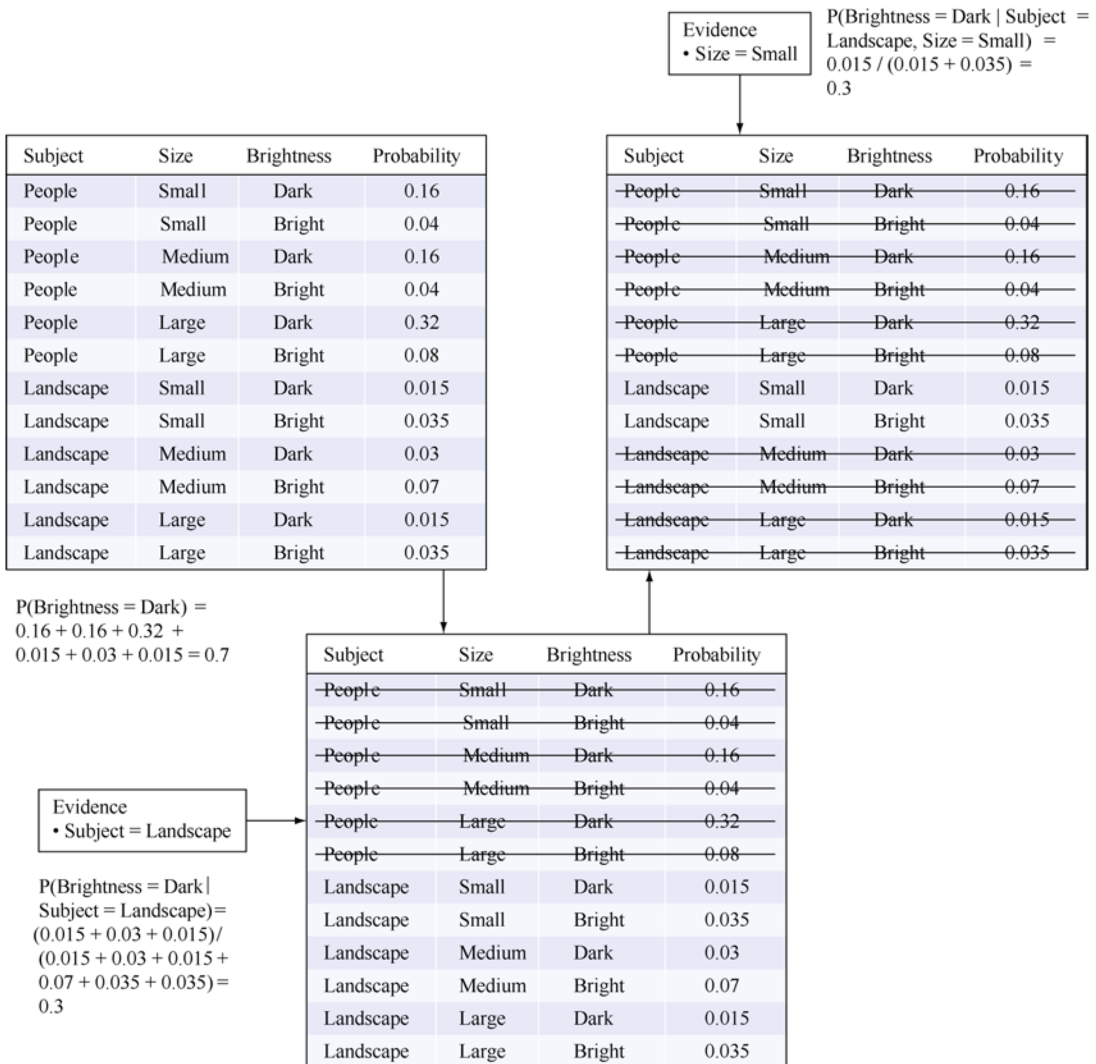
P(Brightness = Dark | Size = Small, Subject = Landscape) =

P(Brightness = Dark | Subject = Landscape)

您可以对 Brightness、Size 和 Subject 的每个可能值得到类似的等式。

要理解这个等式，可以想象图 4-9 中的两步过程。一开始，您对画作一无所知，在变量 Subject、Size 和 Brightness 上有一个先验概率分布。根据这个概率分布，P(Brightness = Dark) = 0.7。此后，您观察到证据 Subject = Landscape，并得到后验概率分布 P(Brightness = Dark | Subject = Landscape) = 0.3。根据我们的先验分布，风景画为深色的可能性低于人物画，所以 Subject = Landscape 这一证据会降低 Brightness = Dark 的置信度。这就是窍门所在：这个后验分布变成下一部分证据的先验分布，现在您观察到 Size = Small，得到一个新的后验分布。根据这个新的分布，P(Brightness = Dark | Size = Small, Subject =

Landscape) = 0.3，所以在知道 Subject = Landscape 的情况下，额外的证据 Size = Small 没有提供关于 Brightness 的任何新信息。如果仔细观察先验分布，就可以发现表格的下半部分中 Subject = Landscape，Dark 和 Bright 的比率始终为 3∶7，而在上半部分 Subject = People，这一比率始终为 4∶1。两个比率不同，但是在固定的 Subject 下，比率是固定的，所以在 Subject 给定的情况下，Size 和 Brightness 是有条件独立的。



Subject	Size	Brightness	Probability
People	Small	Dark	0.16
People	Small	Bright	0.04
People	Medium	Dark	0.16
People	Medium	Bright	0.04
People	Large	Dark	0.32
People	Large	Bright	0.08
Landscape	Small	Dark	0.015
Landscape	Small	Bright	0.035
Landscape	Medium	Dark	0.03
Landscape	Medium	Bright	0.07
Landscape	Large	Dark	0.015
Landscape	Large	Bright	0.035

Subject	Size	Brightness	Probability
~~People~~	~~Small~~	~~Dark~~	~~0.16~~
~~People~~	~~Small~~	~~Bright~~	~~0.04~~
~~People~~	~~Medium~~	~~Dark~~	~~0.16~~
~~People~~	~~Medium~~	~~Bright~~	~~0.04~~
~~People~~	~~Large~~	~~Dark~~	~~0.32~~
~~People~~	~~Large~~	~~Bright~~	~~0.08~~
Landscape	Small	Dark	0.015
Landscape	Small	Bright	0.035
Landscape	Medium	Dark	0.03
Landscape	Medium	Bright	0.07
Landscape	Large	Dark	0.015
Landscape	Large	Bright	0.035

Subject	Size	Brightness	Probability
~~People~~	~~Small~~	~~Dark~~	~~0.16~~
~~People~~	~~Small~~	~~Bright~~	~~0.04~~
~~People~~	~~Medium~~	~~Dark~~	~~0.16~~
~~People~~	~~Medium~~	~~Bright~~	~~0.04~~
~~People~~	~~Large~~	~~Dark~~	~~0.32~~
~~People~~	~~Large~~	~~Bright~~	~~0.08~~
Landscape	Small	Dark	0.015
Landscape	Small	Bright	0.035
~~Landscape~~	~~Medium~~	~~Dark~~	~~0.03~~
~~Landscape~~	~~Medium~~	~~Bright~~	~~0.07~~
~~Landscape~~	~~Large~~	~~Dark~~	~~0.015~~
~~Landscape~~	~~Large~~	~~Bright~~	~~0.035~~

图 4-9 条件独立性：证据 Subject = Landscape 改变 Brightness = Dark 的置信度，但是后续的证据 Size = Small 不提供任何新信息

在概率程序中编码依赖性

第 5 章详细介绍依赖性的主题。该章使用的主要表现形式是贝叶斯网络，我将在此概要介绍。

本书的主题之一是变量之间的依赖性可以编码为一个网络，在这个网络中，如果父变量的值影响子变量的值，则从父变量到子变量存在一个箭头。在我们的例子中，您可以想象画家首先选择主题（Subject），该变量影响尺寸（Size）和鲜艳度（Brightness），但是 Size 和 Brightness 相互之间没有影响。这些关系可以用图 4-10 中描述的网络结构捕捉。简言之，贝叶斯网络中的边是单向的：从父节点指向子节点，它通常表示父变量以某种方式导致子变量的变化。第 5 章提供了这种因果关系的细节以及贝叶斯网络结构的绘制方法。

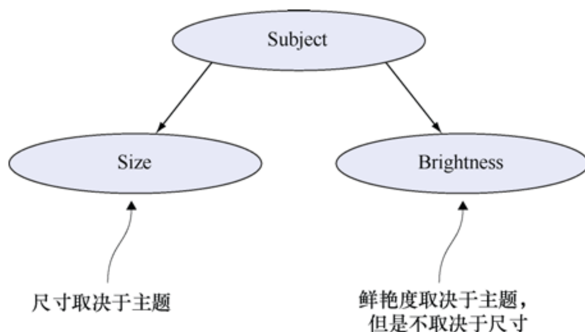


图 4-10 在 Subject 给定的情况下，编码 Size 和 Brightness 之间条件独立性的简单贝叶斯网络结构

以这种风格展示变量间依赖性的网络称作**贝叶斯网络**。网络中的节点对应于模型中的变量，如果第一个变量影响第二个变量，两者之间有一条边。在第 5 章中您将看到，贝叶斯网络中编码了许多关于模型中依赖性的信息。

贝叶斯网络结构自然地转化为概率程序的结构。如果一个变量是网络中另外一个变量的父变量，则第一个变量影响第二个变量。考虑一下常规的程序，如果第二个变量将第一个变量用于自身的定义中，那么第一个变量就会影响第二个变量。概率编程中也是如此：如果贝叶斯网络中的一个变量和另一个变量之间有一条边，则第二个变量将把第一个变量用在自身的定义中。这是概率建模和编程之间的基本联系，它使概率编程成为可能。

理解了这一点，在概率程序中表示依赖性就很简单了。在 Figaro 中，您可以使用如下的程序框架：

```
val subject = // definition
val size = // definition uses subject
val brightness = // definition uses subject
```

重要的是，brightness 的定义中没有使用 size。在第 5 章中您将学习到，这能够捕捉给定 subject 的情况下，brightness 与 size 的条件独立性。

此时，您有了自己的变量，并且了解了用网络形式描述的依赖性。尽管我们已经说过，变量依赖于其他变量，但是您还不了解它们的依赖性以及采用的形式。这通过下面两个成分实现：函数形式和数值参数。

4.3.3 函数形式

好了，您已经选择了变量，并且以网络形式描述了依赖性。我曾经说过，依赖性是用简单组件定义许多变量上的概率分布的关键。这些组件必须仅用关联变量描述变量的特征。下一步是定义描述这些关系特征的组件采取的形式。在图 4-10 的示例中，这一步包括指定 Subject 变量值的选择方式，以及使用该值选择 Size 和 Brightness 变量值的方式。

基本函数形式

我们从第一个问题开始：指定 Subject 值的选择方式。因为 Subject 不依赖于任何其他变量，您所需要做的只是指定 Subject 值的概率分布。在我们的例子中，Subject 采用枚举类型，可取值为 People（人物画）和 Landscape（风景画），所以必须指定 People 和 Landscape 的概率。明确指定概率是表达概率分布的最简方式。但是在有许多可取值或者明确的概率难以直接提出时，这种明确指定可能无法实现。

作为替代，您往往采用一组标准函数形式中的一种指定概率分布。一般来说，可用的形式取决于变量的类型。每种形式都有特定的属性，用于特定的用例。理解不同函数形式的使用方法，以便选择适合于应用的形式，是很重要的。

例如，连续变量的常见函数形式之一是正态分布，另一种是指数分布。对于整数变量，二项和几何分布是两种常见形式。大部分情况下，本书都不过多地讨论单独函数形式的细节。这方面的材料很多，大部分函数形式在维基百科上都有很好的文章。

在第 2 章中已经看到，Figaro 可表现许多常见函数形式。在表现单变量的函数形式时，通常使用原子元素。例如，Select 表现明确概率指定，Normal 表现给定均值和方差下的连续正态分布，Binomial 表现特定试验次数和每次试验成功概率下的二项分布。

条件概率分布

现在，其他两个变量 Size 和 Brightness 是如何依赖 Subject 的？您不能仅仅为它们指定一个基本函数形式，还必须指定它们的概率分布在不同 Subject 值下如何变化。为此，您定义一个条件概率分布（conditional probability distribution，CPD）。您将为 Subject 的每个值定义 Size 上的一个概率分布。例如，您将指定 Subject = Landscape 条件下 Size = Large 的概率，标记为 P(Size = Large | Subject = Landscape)。最简单的方式是指定 Subject 每个可能值对应的 Size 函数形式。

在我们的例子中，Size 是一个枚举变量，所以函数形式自然是明确指定的概率。您可以使用表 4-7 所示的表格呈现给定 Subject 下 Size 的 CPD。我使用 CPD 表格的标准形式，说明 CPD 所属变量以及条件变量。条件变量在左侧，本例中只有一个条件变量 Subject，显示在左列中。您可以想象一个变量依赖于多于一个变量的依赖性模型。在那种情况下，这些变量都显示在左侧。右侧是定义 CPD 的变量，该变量的每个可能值有一列。这里，Size 的 3 个可能值 Small、Medium 和 Large 各有一列。然后，条件变量的每个可能值由 CPD 表中的一行表示。在我们的例子中，条件变量有两个可能值 People 和 Landscape，各由一行表示。如果有不止一个条件变量，这些变量值的每个组合占有一行。最后，在表格中与 *i* 行 *j* 列对应的单元中有一个概率输入项，表示定义了 CPD 的变量在第 *i* 行指定的条件变量值下取得 *j* 列值的概率。例如，在 People 行 Small 列中定义的是 P(Size = Small | Subject = People)。我为这些概率提供了 p1～p6 的标签。这些标签用于后面的代码片段，您可以用它对照表格和代码。

表 4-7　给定 Subject 情况下，Size 的 CPD 形式

Subject	Size		
	Small	Medium	Large
People	p1 = P(Size = Small | Subject = People)	p2 = P(Size = Medium | Subject = People)	p3 = P(Size = Large | Subject = People)
Landscape	p4 = P(Size = Small | Subject = Landscape)	p5 = P(Size = Medium | Subject = Landscape)	p6 = P(Size = Large | Subject = Landscape)

在 Figaro 中，CPD 可以使用复合元素实现。复合元素精确地指定变量依赖于其他变量的方式。第 2 章中，您已经看到 Chain 是用于定义从条件变量值到受制约变量上元素的函数的通用复合元素。这个函数可以是您喜欢的任何函数，所以 Chain 是表示 CPD 的最通用 Figaro 概念。但是还存在许多更特殊的形式。

其中一种有用的形式很自然地称作 CPD。Figaro CPD 形式可以这样应用到我们的例子中：

```
val size = CPD(subject,
  'people -> Select(p1 -> 'small, p2 -> 'medium, p3 -> 'large),
  'landscape -> Select(p4 -> 'small, p5 -> 'medium, p6 -> 'large)
)
```

Scala 变量 p1～p6 表示数值化的概率，您将在下一小节中学到。

注意：引号'表示使用的是 Scala 符号类型。符号是某种事物的特有名称。例如，'small 符号等同于该符号的每次出现，而不同于其他所有符号。除了比较相等之外，对符号不能进行很多其他的处理。当您希望为一个变量创建特定的值集时，符号很有用。

如果您有超过两个变量，也可以使用 CPD。例如，想象有一个 Price（价格）变量与画作是否伦勃朗真迹及其主题有关。可以编写如下形式：

```
val price = CPD(rembrandt, subject,
  (false, 'people) -> Flip(p1),
  (false, 'landscape) -> Flip(p2),
  (true, 'people) -> Flip(p3),
  (true, 'landscape) -> Flip(p4)
)
```

CPD 构造程序有两组参数：

1．**被定义变量的父变量**——最多可以有 5 个父变量。

2．**子句数量**——这个数量根据父变量的可能值而变化。每个子句包含父变量的一种**情况**和一个**结果元素**。例如，在 size 的 CPD 中，第一个子句是情况'people 和结果元素 Select(p1 -> 'small, p2 -> 'medium, p3 -> 'large)。这个子句的含义是，如果 subject 的值为'people，则 size 的值根据 Select 元素进行选择。如果只有一个父变量，情况就是父变量的一个值。如果父变量超过一个，情况就是父变量值的元组。例如，在 price 的 CPD 中，第一个子句的情况是(false, 'people)，也就是说，这个子句适用于 reambrandt 取 false 值、subject 取'people 值的情况。

在 Figaro 中指定依赖性的其他方法

Figaro 提供了另一个构造函数 RichCPD，可以比提供完整表格更简洁的方式表示 CPD。RichCPD 和 CPD 类似，但是每个子句规定每个父变量的一组可能值而不是单一值。您可以三种方式之一指定这些值集：（1）OneOf(values)指定父变量的值必须为给定值之一，该子句才适用；（2）NoneOf(values)规定父变量的值不能为给定的值；（3）*意味着父变量可以取任何值，该子句将适用，假定其他父变量有合适的值。在父变量取一组特定的值时，选中第一个匹配的子句并为元素定义分布。

RichCPD 是更高级的结构，在 CPD 涉及多个父变量时很有用。一般来说，CPD 中的行数等于每个父变量可能取值数量的乘积。当您有 3 个父变量，可能取值分别为 2、3、4 个时，CPD 的行数为 2× 3 × 4 = 24。可以看到，在有更多父变量或者父变量的可能取值更多时，说明这些情况就会变得很乏味。RichCPD 就是设计用于在这些情况下提供帮助的。

下面是一个例子：

```
val x1 = Select(0.1 -> 1, 0.2 -> 2, 0.3 -> 3, 0.4 -> 4)
val x2 = Flip(0.6)
val y = RichCPD(x1, x2,
  (OneOf(1, 2), *) -> Flip(0.1),
  (NoneOf(4), OneOf(false)) -> Flip(0.7),
  (*, *) -> Flip(0.9))
```

有两个父变量的 RichCPD

x1 为 1 或 2，x2 为任何值的匹配情况

x1 为 4 之外的任何值，x2 为 false 的匹配情况

匹配一切的默认情况

警告：CPD 和 RichCPD 中的情况必须覆盖有正概率的所有父变量可能值，这一点很重要。否则，Figaro 不知道遇到没有匹配任何子句的值时该怎么办。对于 CPD，这意味着您必须

提供完整的表格。对于 RichCPD，可以在最后使用“默认”子句，正如上面的例子，对每个父变量使用*。

CPD 和 RichCPD 是编写依赖性的两种明确方法，在使用离散变量时很有用。我不希望您有这样的印象：这是 Figaro 中编写依赖性的唯一方法；它们只是两种简单、明确的方法。在 Figaro 中，指定依赖性的两种通用方法是使用 Apply 和 Chain。回顾第 2 章中的如下要点。

- Apply 取得一系列参数元素和一个函数，其结果是一个元素，该元素的值是将给定函数应用到参数值上的结果。因此，Apply 定义了从参数到结果的依赖性。
- Chain 取得一个参数元素和一个函数，该函数取该参数值并返回一个新元素。Chain 的结果是一个元素，该元素值通过如下途径生成：取得参数值，应用函数以获得新元素，从该元素生成一个数值。因此，Chain 也定义了从参数到结果的依赖性。
- 许多元素（如 If 和复合 Normal）用 Chain 定义。因此，它们也定义了依赖性。

到目前为止，您已经了解了函数形式，并且知道 Figaro 提供了多种函数形式。但是如果没有用于填充其参数的数值，它们就只是个空壳。下面，您将研究这些数值的提供。

4.3.4 数值参数

概率模型中的最后一个成分是数值参数。从概念上讲，这是最简单的部分。您所选择的任何函数形式都有特定的参数，必须为那些参数填入数值。约束之一是必须确保那些值对函数形式是合法的。例如，为一组值指定明确的概率时，这些概率的总和必须为 1。表 4-8 展示了给定 Subject 的情况下填写完成的 Size CPD。注意，每一行的总和都为 1。这是必然的，因为每一行都是 Size 上的一个概率分布。

表 4-8 在 Subject 给定的情况下，填写完整的 Size CPD

Subject	Size		
	Small	Medium	Large
People	0.5	0.4	0.1
Landscape	0.1	0.6	0.3

尽管从概念上很简单，但是填入这些数值可能是实践中最难的步骤。根据专业知识估算概率是一种挑战。回顾本章开头，关于画作的可能趋势有一些综述：伦勃朗“喜欢”在其作品中使用深色；他“常常”创作人物画；新发现的大师真迹很“稀少”，而赝品“多如牛毛”。如果您是某个主题上的专家，提出这样的一般趋势相对容易，但是在这些趋势上加上数字就不同了。伦勃朗“喜欢”在作品中使用深色——是不是意味着 P(Brightness = Dark | Rembrandt = True) = 0.73？很难说。

在这方面，有两个好消息。首先，在许多情况下，得到精确的数字并不关键。即使数字化的概率有所不同，查询的答案通常也大约相同。例如，如果某个模型说明P(Brightness = Dark | Rembrandt = True) = 0.77，您做出的关于画作是否赝品的预测和概率为 0.73 时大约相同。但是，对于接近于 0 的极小概率就不是如此了。将概率从 0.001 改为 0.0001 可能导致结论的彻底改变。在极端条件下，这种数量级的差别很重要。另一方面，专家们有时候更容易提供这样的数量级。

另一个好消息是这些数字通常可以从数据中学习到。您已经在第 3 章中看到了一个例子。值得小心的是，正如那一章中所见，即使从数据中学习参数时，也需要指定一些先验参数。但是这些先验参数可能做出最低限度的假设，在合理的数据量下，这些先验参数的值对最终参数值影响很小。而且，具备指定先验参数值的能力是件好事，因为它提供了使用专业知识影响学习过程的机会。如果您没有专业知识，可以指定中性的先验参数值，不从任何方向影响学习。

在第 9 章和第 12 章中，您将学习到参数学习的基本原理。第 9 章描述了参数学习的两种通用方法。在其中的贝叶斯学习方法中，模型包含了参数值上的一个概率分布。然后，用常规的概率推理实现参数值的学习，生成这些参数值上的后验概率分布。第二种方法不学习参数值上的后验概率分布，而是在看到训练数据之后生成一个估算的参数值。第 12 章说明如何用 Figaro 实现这些方法。

现在，您已经拥有了表现概率模型的实际方法中的全部成分。但是我还没有回答一个重要的问题。假定您用这些成分构建一个概率模型，它如何表现可能世界上的概率分布？下一小节介绍生成模型，这是理解概率编程和程序如何表现概率模型必不可少的一个概念。

4.4 生成过程

本章从“概率模型是可能世界上概率分布的正式表现形式”这一定义开始。4.3 节说明了如何通过指定变量、依赖性、函数形式和数值参数构建概率模型，这一表现方法对于概率编程也是必不可少的。为了清晰地理解概率程序的含义，完成这一闭环，了解这样定义的概率模型如何说明可能世界上的概率分布，是非常重要的。这一理解以**生成过程**的概念为中心。

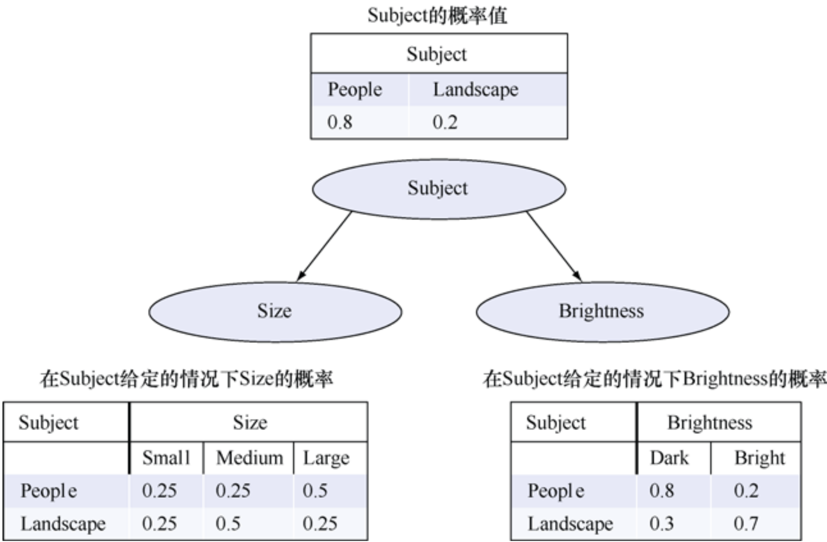
概率程序依赖于概率模型和某种编程语言所编写程序之间的类比。这种类比的实质是：

- 贝叶斯网络等概率模型可以视为对生成变量值过程的定义。这一生成过程定义了可能世界上的概率分布。
- 类似地，常规的非概率程序定义生成变量值的过程。

- 结合上述两者，概率程序使用编程语言描述生成涉及随机选择的变量值生成过程。这一生成过程定义可能世界上的概率分布。

上面的最后一句需要证明，我将解释贝叶斯网络如何通过生成过程定义可能世界上的概率分布，然后将其类推到概率程序。

我们来看看贝叶斯网络如何定义生成变量值的过程。图 4-11 重现了图 4-10 中的贝叶斯网络，以及给定所依赖变量情况下，每个变量的 CPD。



Subject	Size		
	Small	Medium	Large
People	0.25	0.25	0.5
Landscape	0.25	0.5	0.25

Subject	Brightness	
	Dark	Bright
People	0.8	0.2
Landscape	0.3	0.7

图 4-11　包含 CPD 的 3 节点贝叶斯网络

您可以想象生成网络中所有变量值的过程。在已经得到所依赖变量的值之前，您一定不能生成某个变量的值。这一过程如图 4-12 所示。从 Subject 入手，因为它没有依赖任何其他变量。根据 Subject 的 CPD，People 的概率为 0.8，Landscape 的概率为 0.2。所以，您随机选择 People 或者 Landscape，每个都有对应的概率，假设选择的是 Landscape。

然后，选择 Size 变量，您可以这样做是因为已经选择了 Subject 的一个值，这是 Size 变量依赖的唯一变量。因为选择了 Subject = Landscape，所以可以使用 Size 的 CPD 中的对应行。在这一行中，Small 的概率为 0.25，Medium 的概率为 0.5，Large 的概率为 0.25。所以您以这些概率随机选择 Size 的一个值。假设选择 Medium。

最后，为 Brightness 选择一个值，这类似于 Size 值的选择，因为 Brightness 只取决于 Subject，后者的值为 Landscape。假设我们选择的是 Bright。现在，所有变量的生成值是：Subject = Landscape、Size = Medium、 Brightness = Bright。

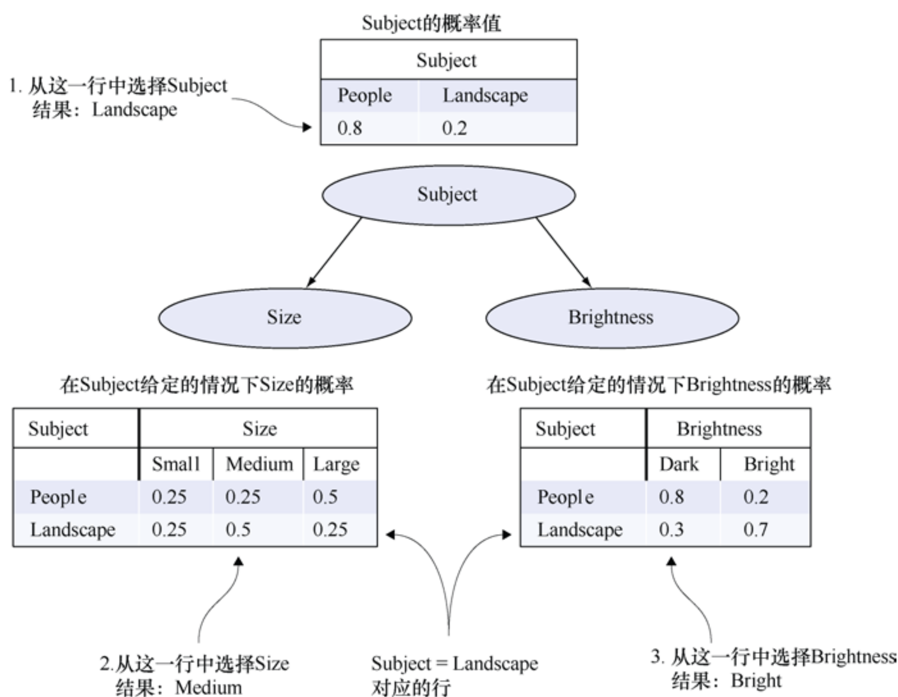


图 4-12　为 3 节点网络的所有变量生成值的过程。变量值通过根据父变量的值，使用对应 CPD 行中的概率选择一个值生成

在经历生成所有变量值的过程时，每个变量都存在多种可能性。可以绘制一棵树捕捉所有可能性。图 4-13 中展示了适合于我们的例子的树。树根是第一个变量 Subject。Subject 的每个可能值表示为一个分支——People 和 Landscape。每个分支用选择的概率注释。例如，根据 Subject 的 CPD，Landscape 的概率为 0.2。然后进入下一个变量 Size，该变量出现在树的两个节点中，对应于 Subject 的可能选择。在每个 Size 节点上，树有 3 个对应于其可能值的分支。注意，与这些分支关联的数字对于两个 Size 节点有所不同。这是因为这些数字来源于 Size 的 CPD 的对应行。根据 Subject 是 People 还是 Landscape，选择的是不同的行。在 Size 节点的每个分支之后是 Brightness 节点，该节点有两个分支，分别对应 Dark 和 Bright，同样，概率取自于 Brightness 的 CPD 的对应行。注意，左侧的 3 个 Brightness 节点概率都相同，右侧的 Brightness 节点也类似。这是因为 Brightness 仅取决于 Subject 而不依赖 Size。左侧的 3 个 Brightness 节点都选择了相同的 Subject 值。

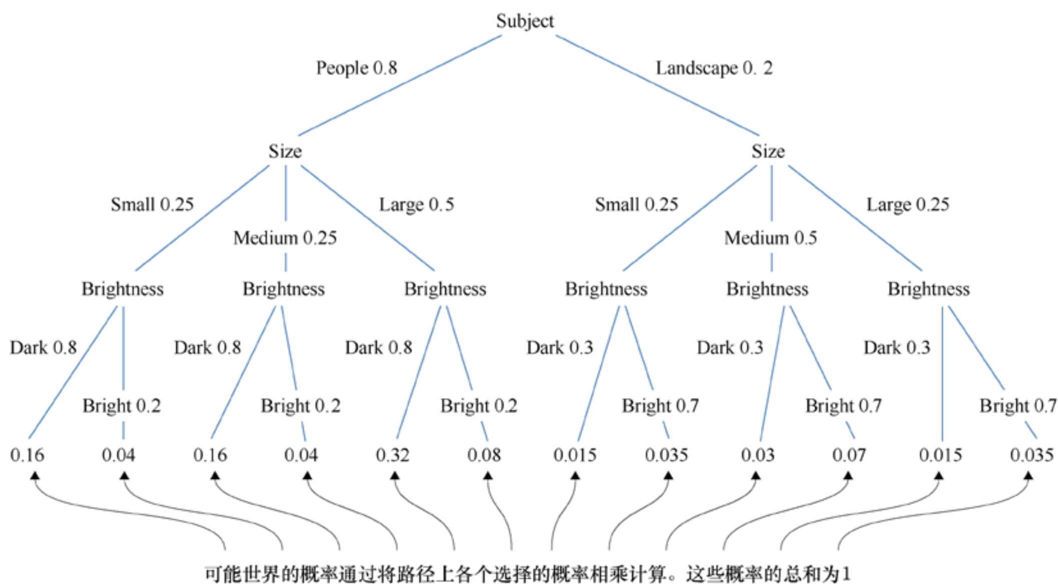


图 4-13 3 节点贝叶斯网络生成概率树

当您经历图 4-13 中的生成过程时，在树中的每个节点都要做出特定的选择。最终得出的选择顺序如图 4-14 所示，您得到了一个特定的可能世界。这个可能世界的概率多大？答案来自于链式法则，这是第 3 部分将会学习的规则之一。根据这一规则，可能世界的概率是生成该世界过程中所做选择概率的乘积。例如，对于可能世界（Subject = Landscape, Size = Medium, Brightness = Bright），概率为 $0.2 \times 0.5 \times 0.7 = 0.07$。证明所有可能世界的概率总和为 1 并不困难。因此，贝叶斯网络定义了可能世界上的概率分布。

现在，您已经了解贝叶斯网络是如何说明生成变量值的过程的。考虑常规编程语言中的一个程序，它也说明了生成变量值的一个过程，只是该过程不涉及随机选择。除了生成变量值的过程涉及随机选择之外，概率程序和常规程序类似。概率程序和贝叶斯网络类似，差别只是使用了编程语言结构描述生成过程。因为您拥有编程语言，就拥有了比贝叶斯网络更多的可能性。您可以使用丰富的控制流（如循环和函数）、复杂的数据结构（如集合和树）。但是原理相同：程序定义了生成所有变量值的过程。

在概率程序的运行中，根据所依赖变量的值，利用定义生成每个变量的值。在程序终止之后，您已经得到一组变量的值。这组变量和生成的值组成了可能世界。正如贝叶斯网络中那样，这个可能世界的概率是每个随机选择概率的乘积。

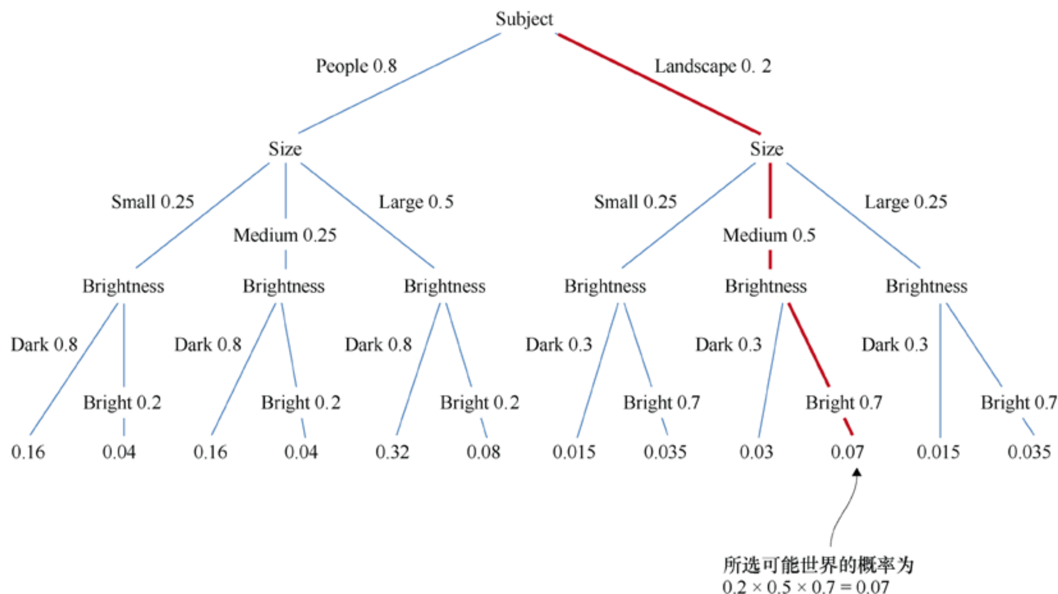


图 4-14 选择顺序及其概率

但是，概率程序和贝叶斯网络之间有很大的差异，每次运行中程序赋值的变量集可能不同。为了理解这一点，考虑如下的 Figaro 程序：

```
Chain(Flip(0.5), (b: Boolean) => if (b) Constant(0.3) else Uniform(0, 1))
```

Constant(0.3)元素只有在 Flip 值为 true 时才会创建，而 Uniform(0, 1)元素只有在 Flip 值为 flase 时才会创建。这两个元素不可能在程序运行中同时创建和赋值。但是，每次运行该程序都会得到一个有明确定义概率的可能世界。而且，尽管证明起来比较难，但是这些可能世界的概率总和仍然为 1。因此，概率程序定义了可能世界上的概率分布。

警告：在编程语言中，编写不终止的程序是有可能的，对于概率编程语言来说也是一样。如果概率程序定义的随机过程存在不终止的可能性，它就不能在可能世界上定义有效的概率分布。想象一下，概率程序定义一个递归函数，不停地调用自身；这一程序定义的过程就不会终止。

到现在为止，您应该已经对概率模型的定义和使用方法以及概率程序如何表现概率模型的所有成分有了很好的认识。在结束本章之前，您需要考虑另一个技术要点——如何在概率模型中使用连续变量。

4.5 使用连续变量的模型

到目前为止，您所看到的概率模型示例使用的是离散变量。离散变量（如枚举和整数）的值有明确的分隔，而连续元素（如实数）的值没有分隔。第 2 章提示了连续变量的处理不同于离散变量这一事实，并承诺将提供完整的解释。下面就是这个解释。

连续元素带来的挑战是，难以在所有连续元素之上定义概率分布。我们来考虑一下任何实数区间。如果对区间中的所有数值指定正概率，总概率将为无穷大。不管区间多小，这都是事实。那么，如何定义连续变量的概率呢？本节提供这个问题的答案。

为了具体说明，我将使用一个β-二项式模型。这种使用的模型有许多应用。在第 3 章中，您已经使用这个模型表示电子邮件中的不寻常单词数量。本节详细介绍β-二项式模型，以说明本章目前为止学习的概念。这个模型包含一个连续变量，我将解释如何表现这个变量，并解释连续变量的一般用法。

4.5.1 使用β-二项式模型

β-二项式模型这一名称的由来是因为该模型结合使用了β-分布和二项分布。我们来看看在这个简单的示例中，这些分布出现在什么位置。假定您有一枚不均匀的硬币，其正面与背面的概率不同。您的目标是根据前几次掷硬币的观察，预测掷出正面的概率。想象一下，您观察 100 次掷币的结果，希望查询第 101 次掷出正面的概率。

首先定义变量。对于这个例子，您将使用如下 3 个变量。

- 硬币的偏差，称作 Bias。
- 前 100 次掷币掷出正面的次数，称作 NumberOfHeads，这是一个取值为 0 到 100 之间的整数。
- 第 101 次掷币的结果，称作 $Toss_{101}$。

至于依赖性模型，使用图 4-15 中的贝叶斯网络。记住，尽管您从正面向上的次数推断偏差，但是模型中的依赖性方向可能与推理相反。从因果关系上看，硬币的偏差决定了每次掷币的结果，相应地驱动正面向上的次数，所以边从 Bias 指向 NumberOfHeads。

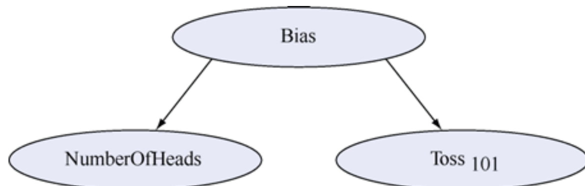


图 4-15 掷币示例的贝叶斯网络。前 100 次掷币用一个表示正面向上次数的变量加以总结

您可能觉得疑惑，为什么没有从 NumberOfHeads 到 $Toss_{101}$ 的边。毕竟，前 100 次掷币中正面向上的次数提供了与第 101 次掷币相关的信息。原因是，所有信息是以硬币的偏差为媒介的。前 100 次掷币告诉您关于硬币偏差的信息，然后用这些信息预测 $Toss_{101}$。如果您已经知道偏差，前 100 次掷币就没有提供任何附加信息。换言之，在 Bias 给定的情况下，$Toss_{101}$ 条件独立于 NumberOfHeads。

现在，您可以选择函数形式。每个变量有一个函数形式，我们从 $Toss_{101}$ 开始。这是一次掷币的结果，不是正面就是背面，是最简单的形式之一，只需要指定正面或者背面的概率。在我们的例子中，正面向上的概率由 Bias 变量值给出，与 Bias 相等，而背面朝上的概率等于 1-Bias。在 Figaro 中，可以使用一个复合 Flip，其中的概率由偏差给出。

接下来，您需要一个函数形式，描述在每次掷币结果的概率与 Bias 值相等的情况下，总共掷币 100 次时，掷出正面的特定次数的概率。在第 2 章中已经看到，对于这种情况，合适的函数形式是二项分布。在二项分布中，您重复进行可能得到两个结果之一的相同试验。这种分布描述了某种结果出现次数的概率分布。在我们的例子中，它将告诉我们观察 100 次掷币时出现特定次数正面的概率。

一般来说，二项分布取决于两个参数：试验次数（我们的例子中是 100）以及任何特定试验取得预期结果（在我们的例子中是硬币的正面向上）的概率。如果您知道这个概率，就可以立即计算出特定次数（如 63 次）正面向上的概率。但是，在我们的例子中，这个概率由 Bias 变量值给出。

您需要 Bias 的函数形式。从模型名称中，您可能已经猜到，将要为偏差使用β分布。一般来说，β-二项式模型指的是包含表示试验成功次数的二项变量，成功概率由β变量指出的模型。

您已经知道硬币的偏差呈β分布。但是，这意味着什么呢？如何得到连续变量上的概率分布？

4.5.2　连续变量的表示

Bias 的值是 0～1 的实数，所以它是一个连续变量。连续变量需要不同的方法指定概率分布。例如，假定您认为一幅画的高度在 50 到 150 厘米之间，且不认为任何一个高度的可能性超过其他高度。因此，其值域为 50～150 厘米，您相信该范围内的概率是均匀的。您可能想要知道高度恰为 113.296 厘米的概率，问题是可能的高度有无穷多个。如果任何高度有正概率（如 0.2），且您相信所有高度都有相同的概率，那么就有无穷多个概率为 0.2 的高度，50 和 150 厘米之间的所有高度的总概率将为无穷大。但是，您知道总概率必须为 1。

解决方案是不考虑单独高度的概率，而是观察高度的一个**区间**或者范围。例如，如

果查问高度在 113 厘米到 114 厘米之间的概率，您可以说是 1/100，因为这是在 100 个相同可能性的厘米数中的一个。实际上，您可以查询 113.296 厘米周围任意区间的概率，不管这个区间有多小。例如，[113.2，113.3]区间的概率为 1/1000；[113.29，113.30]区间的概率为 1/10000，依此类推。

因此，对于连续变量，单独的点概率通常为 0。我们来考虑硬币的偏差，偏差恰为 0.49326486…的概率为 0，因为还有无穷多个其他点。但是包含这些点的区间有正概率。例如，偏差可能在 0.49326 和 0.49327 之间。

图 4-16 描述了指定每个区间概率的一种数学方法——使用概率密度函数（PDF）。PDF 是指定每个区间概率的一种方法，它通过提供每个点的密度起作用。如果区间很小，密度在整个区间上可能大致相同，所以该区间的概率等于恒定的概率乘以区间的长度。在图 4-16 中显示的是均匀 PDF，密度完全固定在值 0.01，所以区间（113,114）的概率等于 0.01 乘以区间长度 1，也就是 0.01。当密度函数完全平直时，将恒定密度乘以区间长度可以得出该区间密度曲线之下的面积。一般来说，密度在不同点可能变化，在这种情况下，区间概率为曲线之下面积的原理也同样适用。如果您还记得微积分知识，就会知道区间中曲线之下的面积等于密度函数在该区间内的积分。本书中不要求您进行任何微积分计算，或者计算任何曲线下的面积——Figaro 负责这项工作，但是我希望确保您能理解，连续分布与其他分布略有不同，并且确保您知道密度函数是什么。

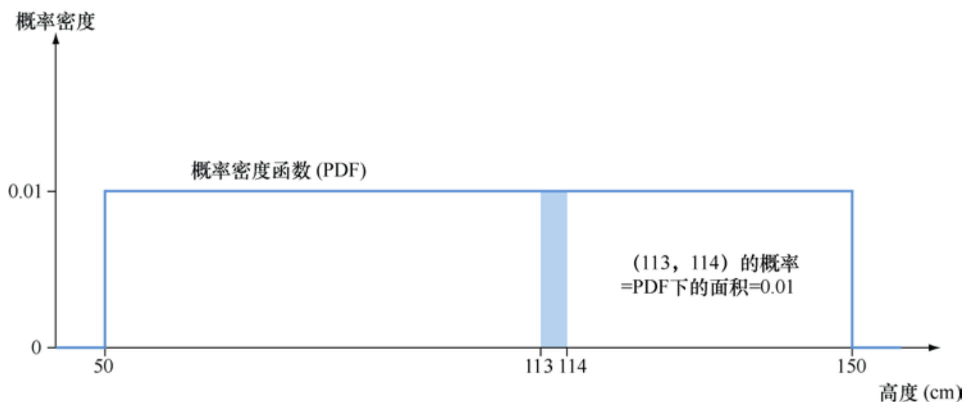


图 4-16　概率密度函数（PDF）：区间的概率等于该区间中 PDF 曲线下区域的面积
（此图没有按照比例绘制）

回到我们的例子，您将为 Bias 寻找一个函数形式，需要为其指定一个 PDF。您知道偏差是一个概率，因此必然在 0 和 1 之间，所以在（0,1）区间之外的密度必然为 0。有一个函数形式正好适合于这种情况，这就是β分布。β分布用于建立有两种可能结果的试验产生某种结果的概率模型。图 4-17 展示了β分布的一个例子。可以看到，这种分布只在 0 和 1 之间有正概率，有单一的波峰。很快您就会看到如何描述峰值所在以及图形

的尖锐程度。

选择β分布描述硬币偏差特征的原因是β分布和二项分布结合得很好。具体地说，如果有关于 NumberOfHeads 的证据且 Bias 的先验分布是β分布，那么在看到硬币正面朝上次数的证据之后，Bias 的后验分布也是β分布。您将在第 3 部分中看到，以这种方式预测第 101 次掷币结果很简单。

总结起来，目前为止您已经拥有了如下变量及其函数形式。

- Bias，由β分布描述其特性。
- NumberOfHeads，由总试验数量为 100，每次试验中正面向上概率为 Bias 值的二项分布描述。
- $Toss_{101}$，由 Flip 描述，其中正面的概率等于 Bias 的值。

下面是模型的 Figaro 框架：

```
val bias = Beta(?,?)
val numberOfHeads = Binomial(100, bias)
val toss101 = Flip(bias)
```

现在进行描述模型的第 4 步：选择数值参数。观察程序框架，可以看到 numberOfHeads 和 toss101 的相关参数由 bias 的值指定，后者已经是程序中的一个元素。您只需要指定 bias 的参数即可。为此，您必须理解β分布的参数。

β分布有许多版本，图 4-17 中展示了一个特殊的版本。每个版本的特性由两个数值参数α和β描述。图中的版本是 beta(2, 5)，即α值为 2，β值为 5。

这两个参数有一种自然的解读，代表观察到各种结果的次数加 1。在我们的例子中，α是已经观察到的硬币正面向上次数加 1，而β则是背面向上的次数加 1。因此，在观察任何掷硬币的试验之前，先验α和β参数表示您认为在想象的掷硬币试验中观察到的结果。如果您认为自己对此一无所知，没有想象任何掷硬币试验，可以将α和β都设置为 1。如果根据类似硬币的经验认为这个硬币正面比背面更容易出现，可以将α设置为大于β。对这一经验越有信心，α和β的量级就越大。

例如在图 4-17 中，α为 2，β为 5。因为β大于α，您的先验信念倾向于该硬币偏向背面（正面的概率较小），确实，可以看到低偏差值（<0.5）的概率密度高于高偏差值（>0.5）。而且，从α= 2 和β= 5 开始，暗示着您已经看到了一次正面和四次背面，所以将从一些关于偏差的先验知识入手，PDF 不是直线正反映了这一点。

您已经看到了概率模型的全部 4 种成分，以及在概率程序中如何表示它们。现在，您已经为深入建模技术和设计模式做好了准备。下一章将从依赖性的详细讨论开始，并介绍两种描述它们的代表性框架——贝叶斯网络和马尔科夫网络。

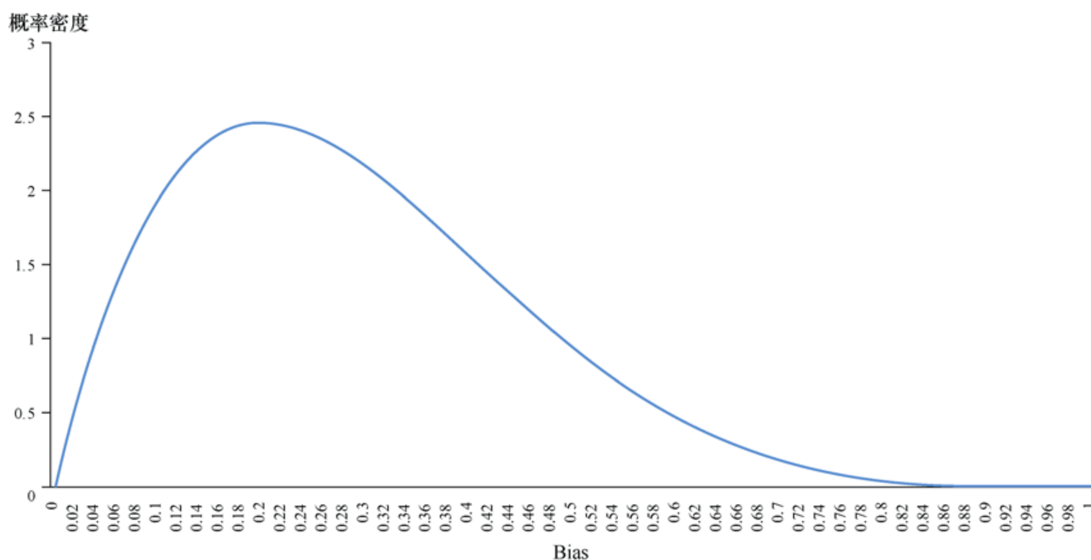


图 4-17　beta(2, 5)分布的 PDF

4.6　小结

- 概率模型以可能世界上概率分布的形式表示一般领域知识。
- 您使用证据排除与证据不一致的可能世界，并规格化其余可能世界的概率。概率编程系统使用推理算法负责这一工作。
- 构建概率模型需要指定变量及其类型、网络形式的变量间依赖关系以及每个变量的函数形式及数值参数。
- 连续变量的函数形式由一个概率密度函数指定。
- 概率模型的成分定义生成所有变量值的过程。
- 概率程序使用编程语言定义生成过程。

4.7　练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．考虑一个 5 张牌的扑克游戏：黑桃 A、黑桃 K、红心 K、黑桃 Q 和红心 Q。每个游戏者得到一张牌。可能世界有哪些？假定牌已经洗得很好，每个可能世界的概率是多大？

2．在同一个扑克游戏中，您观察到一位游戏者有一张人头牌（K 或者 Q）。其他游戏者有一张黑桃的概率多大?

3．我们让扑克游戏的规则更复杂一些。游戏包括下注，下面是下注的规则：

游戏者 1 可以下注或者通过。

如果游戏者 1 下注：

游戏者 2 可以下注或者放弃。

如果游戏者 1 通过：

游戏者 2 可以下注或者通过。

如果游戏者 2 下注：

游戏者 1 可以下注或者放弃。

可能的下注结果是：（1）两位游戏者都下注；（2）两位游戏者都通过；（3）其中一位下注，另一位通过。如果两者都通过，则他们都没有任何输赢。如果一位下注，另一位放弃，则下注的游戏者赢得 1 美元。如果两者都下注，牌较大的游戏者赢得 2 美元。如果两位游戏者的牌同级，黑桃战胜红心。

您打算为这个扑克游戏构建一个概率模型。模型的目标是根据每次行动发生的结果，帮助您决定游戏中任何时点的行动。

a）模型中有哪些变量?

b）变量之间有何依赖性?

c）这些依赖性的函数形式是什么?

d）您确定的数值参数有哪些？哪些参数必须估算或者从经验中学习?

4．编写一个 Figaro 程序表示游戏的概率模型。假定某些参数值必须估算。使用程序做出如下决定：

a）您是游戏者 1，得到了一张黑桃 K。应该下注还是通过?

b）您是游戏者 2，得到了一张红心 K，对手下注。您应该下注还是放弃?

c）现在，看看能否改变估算参数值，改变决定。

5．通过练习 4 中的程序生成过程，描述 3 种不同的选择顺序。使用程序第 1 版本的参数，在没有观察证据的情况下，每个顺序的概率是多大?

6．您怀疑对手发牌时不公正。具体地说，您相信她以未知的概率 p 发给您红心 Q，而在其余情况下（概率为 $1-p$），她以随机的方式发牌。您相信概率 p 遵循 Beta（1,20）分布。在您得到的 100 张牌中，有 40 张是红心 Q。使用 Figaro 计算下一张牌是红心 Q 的概率。

第 5 章　用贝叶斯和马尔科夫网络建立依赖性模型

本章介绍如下内容：

- 概率模型中变量之间的关系以及这些关系如何转化为依赖性
- 如何用 Figaro 表达不同类型的依赖性
- 贝叶斯网络：编码变量之间有向依赖性的模型
- 马尔科夫网络：编码变量之间无向依赖性的模型
- 贝叶斯和马尔科夫网络的实际示例

在第 4 章中，您学习了有关概率模型和概率程序间关系的知识，还了解了概率模型的成分——变量、依赖性、函数形式和数值参数。本章的焦点是两种建模框架：贝叶斯网络和马尔科夫网络。这两种框架基于不同的依赖性编码方法。

依赖性捕捉变量之间的关系。理解关系的种类和如何转化为概率模型中的依赖性，是构建模型中所能得到的最重要技能之一。相应地，您将学习关于各类关系及依赖性的知识。一般来说，变量之间有两类依赖性：有向依赖性表达不对称关系，无向依赖性表达对称关系。贝叶斯网络编码有向依赖性，而马尔科夫网络编码无向依赖性。您将学习如何用编程语言的能力扩展传统贝叶斯网络，从概率编程的能力中获益。

理解本章中的材料之后，您将对概率编程的精髓有扎实的知识。所有概率模型都可以归结为一组有向和无向依赖性，您将知道何时引入变量之间的依赖性，依赖性是有向还是无向，如果有向，应该使用哪一种方向。第 6 章以这些知识为基础，用数据结构创建更复杂的模型，第 7 章将进一步用面向对象建模方法扩展您的技能。

本章假定您已经有了 Figaro 的基本知识，这些知识在第 2 章中已经出现。特别是，您应该熟悉了有向依赖性的基础 Chain，以及无向依赖性的基础——条件与约束。您还将使用第 4 章中出现过的 CPD 和 RichCPD 元素。如果不记得这些概念也不用担心，我将在您看到它们的时候提醒。

5.1　建立依赖性模型

概率推理是关于变量间依赖性运用的技术。如果一个变量的知识为其他变量提供信息，则两个变量是**相关**的。相反，如果知道一个变量的某些信息不能告诉您关于其他变量的任何信息，这两个变量是**独立**的。

考虑一个计算机系统诊断应用，在这个应用中您尝试推理某个打印进程的故障。假定有两个变量：Printer Power Button On（打印机电源按钮开启）和 Printer State（打印机状态）。如果观察到电源按钮关闭，可以推断打印机状态为关闭。相反地，如果观察到打印机状态关闭，也可以推断电源按钮可能关闭。这两个变量明显相关。

依赖性用于建立以某种方式相关的变量的模型。变量之间存在许多类关系，但是依赖性只有两类：

- **有向依赖性**从一个变量指向另一个变量。通常，这些关系建立变量因果关系的模型。
- **无向依赖性**建立两者之间影响没有显著方向的变量关系模型。

下面两个小节详细描述这两类依赖性，并提供丰富的示例。

5.1.1　有向依赖性

有向依赖性从一个变量指向另一个变量，通常用于表示因果关系。例如，打印机的电源按钮关闭导致打印机停机，所以 Printer Power Button On 和 Printer State 之间存在有向依赖性。图 5-1 说明了这种依赖性，两个变量之间有一条有向边。（**边**是表示图中两个节点之间箭头的常用术语）第一个变量（这里是 Printer Power Button On）称为 **父变量**，第二个变量（Printer State）称作**子变量**。

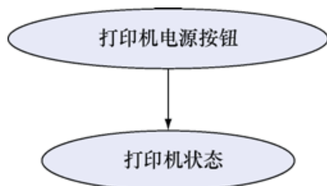


图 5-1 表示因果关系的有向依赖性

为什么箭头从因到果？理由很简单，因通常发生在果之前。更深入的答案与第 4 章探索过的生成模型概念有关。您应该记得，生成模型描述了模型中所有变量值的生成过程。一般来说，生成过程模拟某种现实世界的过程。如果某个原因导致某种结果，您应该首先生成原因的值，并用该值生成结果。在我们的例子中，如果创建一个打印机模型，并且想象为模型中的所有变量生成值，您将首先生成 Printer Power Button On 的值，然后生成 Printer State 值。

现在，值得重申的是，**依赖性的方向不一定是推理的方向**。您可以从打印机电源按钮关闭推断出打印机状态为停机，但是也可以从相反的方向推理：如果打印机状态为正常运行，可以确定打印机按钮没有关闭。许多人错误地按照预想的推理方向构造模型。在诊断应用中，您可能观察到打印机停机并且试图确定其原因，所以从打印机状态推断出打印机电源按钮是否开启。您可能有绘制一条从打印机状态到打印机电源按钮开启的箭头的冲动。这是错误的。箭头应该表达生成过程，遵循因果关系的方向。

我曾经说过，有向依赖性通常建立的是因果关系模型。实际上，因果关系只是变量间不对称关系的一个例子。下面我们更仔细地观察各种类型的不对称关系——首先是因果关系，然后是其他类型。

各种因果关系

下面是因果关系的一些种类。

- **先发生的情况与接下来发生的情况**——最明显的因果关系是某种情况导致后来发生的另一种情况。例如，如果某人关闭打印机电源，此后打印机将停机。这种时间关系是因果关系的常见特性，以至于您可能认为所有因果关系都涉及时间，但是我不同意这一点。
- **状态的因果关系**——有时候，您可能用两个变量代表某一时刻状态的不同方面。例如，可能用一个变量表示打印机电源按钮是否关闭，另一个变量表示打印机是否停机。两者都是同一时刻的状态。在这个例子中，打印机按钮关闭导致打印机停机，因为它使打印机失去电源供应。

- **真实值与计量值**——每当一个变量是另一个变量值的**计量**，您就会说真值是计量值的根源。例如，假定您有一个 Power Indicator Lit 变量，表示打印机电源 LED 是否点亮。Printer Power Button On 到 Power Indicator Lit 存在不对称关系。通常，计量值由传感器产生，同一个值可能有超过一个计量。而且，计量值通常是通过观测而得，您希望从计量值推断真值，所以这是依赖性方向不同于推理方向的另一个例子。
- **参数和使用参数的变量**——例如，考虑硬币的偏差（表示掷币出现正面的概率）和掷币结果。掷币使用偏差确定结果。很明显，偏差是先生成的，然后才是单独的掷币结果。在多次掷同一枚硬币时，它们都是在偏差之后生成的。

其他不对称关系

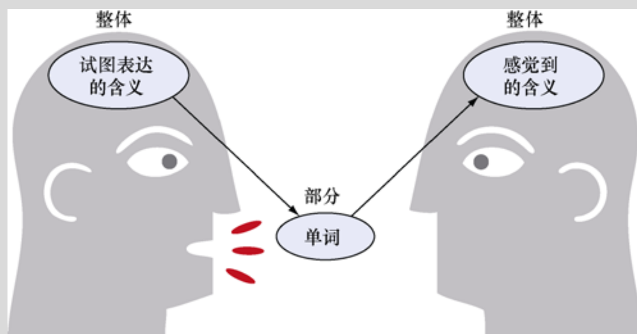
到目前为止，前面介绍的情况都是最重要、最明确的。如果理解了这些情况，就能够在 95%的情况中确定依赖性的正确方向。现在，我们更进一步，考虑其他的各种关系，这些关系虽然明显是不对称的，但是在依赖性方向上不明确。我将列出这些关系，然后描述帮助您解决歧义的经验法则。

- **部分到整体**——对象中某个部分的属性往往决定整体属性。例如，考虑一台有硒鼓和进纸器的打印机。作为打印机的部件，硒鼓或者进纸器的故障都可能导致整台打印机的故障。其他一些时候，整体的属性可能决定某个部分的属性。例如，如果打印机整体制作工艺低下，进纸器和硒鼓也很有可能质量不佳。
- **特殊到一般**——这种关系也可能是双向的。用户可能以许多方式遭遇打印机问题，如卡纸或者打印质量低下。如果用户遇到这些特殊问题，也就将遇到较为一般的打印机体验问题。在这种情况下，特殊问题导致了一般的问题。另一方面，想象一下生成对象的过程。通常，您在用细节精练对象之前生成一般属性。例如，生成一台打印机时，您可能首先决定它是激光打印机还是喷墨打印机，然后才生成单独的属性。确实，在您知道打印机的类型之前，生成可能仅与特定类型有关的特殊属性没有意义。
- **具体/详细到抽象/概要**——具体-抽象关系的例子是考试成绩与字母级别之间的关系。许多成绩对应于相同的字母级别。很明显，教师根据考试成绩而不是其他方式决定字母级别，所以测试成绩导致字母级别。另一方面，考虑学生和考试成绩的生成过程。您可能首先生成抽象的学生类型（如 A 学生或者 B 学生），然后生成具体的考试成绩。

消除因果关系的歧义

从前面的例子中可以看到，尽管这些情况中存在明显的不对称依赖性，但是梳理依赖性的方向可能很困难。下面的思路可能有所帮助。想象某人正在用英语向另一个人讲一句话，如下

图所示。说话者对整句话的含义心中有数，从这个含义中，说话者生成了各个单词。这是一个整体–部分关系，方向是从句子到单词。然后，其他人听到了这个句子。这个人将单词的含义组合起来，创建了一个部分–整体关系。



句子的传达：说话者试图表达的整句话含义生成单词（部分），
这些单词生成了听者所感觉到的整句含义

仔细观察这个例子，就可以看到造句的过程，整个句子的含义在单独单词之前产生。但是在理解句子的过程中，句子的含义是在单独的单词之后感觉到的。这不是一条铁律，但是在您制作某种事物的时候，首先会生成一般/抽象/整体，然后加以精练，产生特殊/具体/详细/多部分的产品。当您生成打印机时，首先生成打印机的一般分类。然后生成打印机的具体型号，以及关于打印机各个部件的详细信息。类似地，在生成学生时，首先生成学生的抽象分类，然后填入具体的考试成绩。另一方面，当您理解和报告某个事物时，首先感受到的是关于其各个部分的特殊/具体/详细信息，然后衍生和总结关于整体的一般/抽象属性。例如，遭遇特殊打印机问题的用户可能以一般的方式总结，给学生评定级别的教师可能首先观察考试成绩，然后报告字母级别。所以，经验法则如下。

如果您打算建立属性定义的模型，依赖性从一般、抽象或者整体概念指向特殊、具体或者部分。如果打算建立属性理解和报告的模型，依赖性从特殊、具体或者部分指向一般、抽象或者整体。

如前所述，如果您理解了主要的因果关系，几乎在任何时候都能发现正确的方向。除了这个规则之外，即使有经验的建模人员在依赖性的正确方向上都无法达成一致。我希望这个经验法则能够帮助您构建模型。

Figaro 中的有向依赖性

记得概率模型的 4 种成分吗？它们是变量、依赖性、函数形式和数值参数。到目前为止，您假定变量是以依赖性为中心的。当您希望在 Figaro 中表达这些依赖性时，需要提供一个函数形式并指定数值参数。

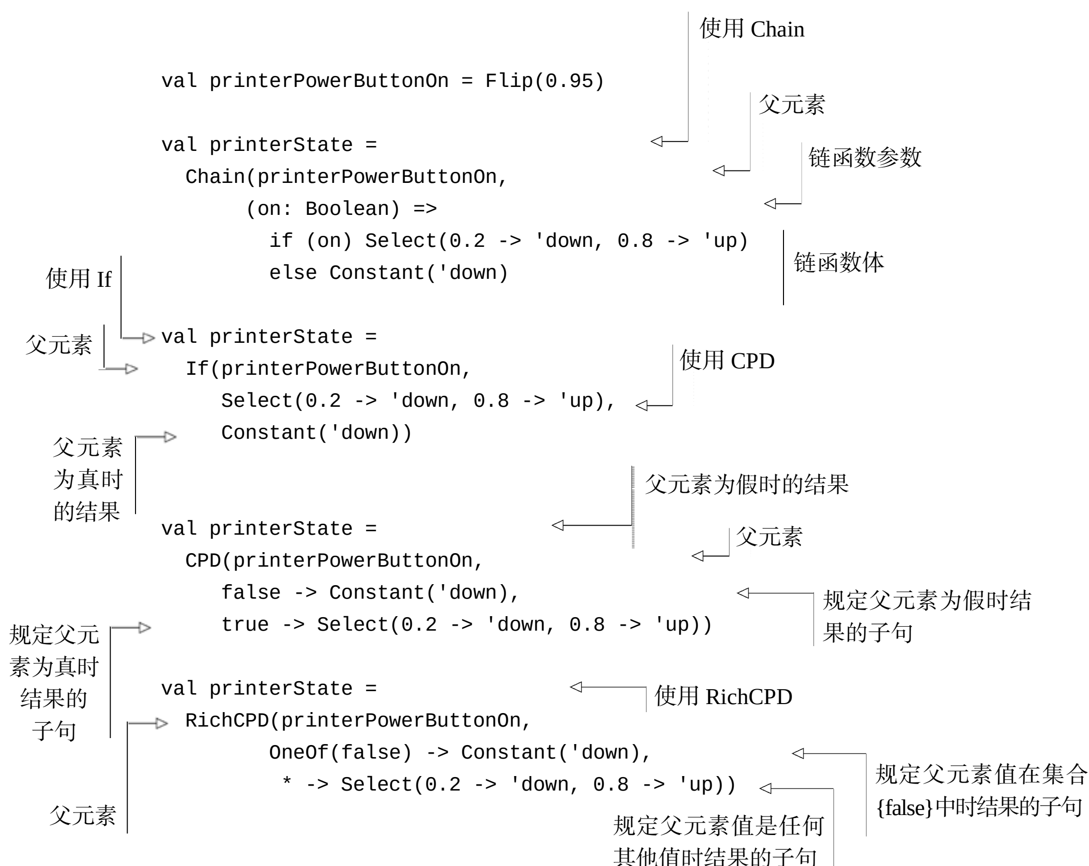
您可以用多种方法在 Figaro 中表达有向依赖性。一般的原则是使用某种 Chain 元素作为函数形式。Chain 有两个参数：

- 父元素。
- 从父元素值到结果元素值的链函数。

Chain(parentElement, chainFunction)定义如下的生成过程：从 parentElement 获得父元素的一个值，然后应用 chainFunction 得到 resultElement，最后从 resultElement 得到一个值。

用 Chain 表达有向依赖性时，parentElement 自然是父元素。链函数指定对于每个父元素值，子元素上的概率分布。Figaro 有许多结构使用 Chain，所以在许多情况下，尽管看上去不像，但是使用的仍然是 Chain。

下面是一些等价的例子，它们都表示，如果打印机电源按钮关闭，打印机绝对会停机，但是如果电源按钮打开，打印机可能正常运行，也可能停机。



如前所述，有一类非对称关系出现在参数和依赖该参数的变量之间，如硬币的偏差和掷币结果。同样，这可以使用 Chain 或者原子元素的复合版本表达。下面是两个等价的例子：

```
val toss = Chain(bias, (d: Double) => Flip(d))

val toss = Flip(bias)
```

5.1.2　无向依赖性

您已经了解，有向依赖性可以表示各种不对称关系。无向依赖性建立变量之间无明显方向关系的模型。这种关系称作**对称关系**。如果您的变量相互关联，但是没有一个变量在另一个变量之前生成的明显生成过程，那么无向依赖性可能更合适。对称关系可能以两种形式出现。

- **相同原因产生的两种结果，原因没有明确的模型**——例如，同一个值的两种计量且您没有用于该值的变量时，或者相同事件的两种结果且没有用于该事件的变量时。很明显，如果您不知道事件的潜在值，这两种计量或者结果是相关的。想象一下，在打印机的例子中，您有两个不同的变量——打印质量和打印速度。如果没有一个表示打印机状态的变量，这两个变量是相关的，因为它们是可能有相同潜在根源的打印事务的两个方面。

  您可能会问，为什么我们不在模型中包含用于根源的变量？答案可能是，这个根源比结果复杂得多，难以精确建模。在本章中，您将看到一个图像重现的例子。图像是复杂三维场景的二维效果。创建三维场景的正确概率模型，可能比建立图像中像素间关系的模型要困难得多。
- **同一个已知结果的两个原因**——这种关系很有趣。通常，同一结果的两个原因之间没有关系。例如，打印机的进纸器状态和硒鼓墨粉水平都影响打印机的整体状态，但是进纸器状态和墨粉水平是相互独立的。不过，如果您知道打印机打印效果不佳，进纸器的状态和墨粉水平突然变成相关的。如果知道墨粉不足，就可能导致您认为打印质量不佳的原因是墨粉不足而不是进纸不畅。在这个例子中，打印机的整体状态是结果，而墨粉水平和进纸器状态是可能的原因，在结果已知的情况下，这两个原因就变成相关的。这是**诱导依赖性**的一个例子，在 5.2.3 小节中您将更详细地学习这种依赖性。如果您没有用于该结果的变量，这就变成了两个原因之间的对称关系。但是将两个原因造成的共同结果排除在模型之外的情况不太多见。

在 Figaro 中表达无向依赖性

在 Figaro 中，您可以用两种方法表达对称关系：约束和条件。这两种方法各有利弊。

约束方法的好处是概念较为简单。但是约束中出现的数字是硬编码的，学习算法无法访问，因此不能在 Figaro 中学习到。使用条件方法的好处是数值可以学习。

两种方法的基本原理类似。5.5 节详细描述了无向依赖性的编码方法，但是这里有一个简单的版本。当两个变量之间存在某种无向依赖性时，两个变量值的某些组合比其他组合更常见。这可以通过对不同组合指定权重实现。约束通过指定两个变量值的某个组合到表达该值权重的实数之间的一个函数，编码了这些权重。在条件方法中实际上编码了相同信息，但是采用了更复杂的方式。

例如，我们编码图像中两个相邻像素之间的关系。这些关系有"其他所有条件都相同"的特性。例如，您可能认为，在其他条件都相同的情况下，两个像素颜色相同的概率 3 倍于颜色不同的概率。实际上，许多关系可能影响两个像素的颜色，所以它们有相同颜色的概率并没有真的达到 3 倍。

下面是在 Figaro 中用约束方法表达这种关系的方式。我们将两个像素的颜色称作 color1 和 color2。为了例子的简单，您将假定颜色是布尔变量。记住，约束是从一个元素值到双精度类型值的函数。您必须创建一个表示 color1 和 color2 配对的元素，以便对该配对指定约束。代码如下：

```
import com.cra.figaro.library.compound.^^
val pair = ^^(color1, color2)
```

^^是 Figaro 配对构造程序

现在，您必须定义实现约束的函数：

```
def sameColorConstraint(pair: (Boolean, Boolean)) =
  if (pair._1 == pair._2) 0.3; else 0.1
```

这个测试检查配对的第一部分是否等于第二部分

最后，对颜色配对应用约束：

```
pair.setConstraint(sameColorConstraint _)
```

下划线表示您想要的是函数本身，而不是应用它

Figaro 对这个约束的解释正如您的预想。在其他条件相同的情况下，配对的两个部分相等（换言之，两个颜色相同）的概率 3 倍于两者不同的概率。正如 5.5 小节中的定义，这些约束共同地以正确的方式定义了概率分布。但是很遗憾的是，数值 0.3 和 0.1 埋藏在约束函数内，所以它们无法从数据中学习。

对于条件方法，我将展示代码，然后解释其正确工作的原因。首先，定义一个辅助的布尔型元素，称作 sameColorConstraintValue。然后定义其为 true 的概率为约束的值。最后，添加一个条件，规定这个辅助元素必须为 true。这可以用一个观测值实现：

```
val sameColorConstraintValue =
  Chain(color1, color2,
        (b1: Boolean, b2: Boolean) =>
          if (b1 == b2) Flip(0.3); else Flip(0.1))
sameColorConstraintValue.observe(true)
```

这段代码与约束的版本等价。要明白这一点，就需要知道条件导致任何违反条件的值取概率 0；否则概率为 1。任何状态的概率通过组合来自元素定义和条件或约束的概率得到。例如，假定两个颜色相同。使用 Chain 的定义，sameColorConstraintValue 为 true 的概率是 0.3,为 false 的概率是 0.7。如果 sameColorConstraintValue 为 true，条件的概率为 1，而如果 sameColorConstraintValue 为 false，条件的概率为 0。因此，这个条件的组合概率为(0.3 × 1) + (0.7 × 0) = 0.3。类似地，如果两个颜色不相同，组合概率为(0.1 × 1) + (0.9 × 0) = 0.1。可以看到，在其他条件都相同的情况下，颜色相同的可能性 3 倍于颜色不同的可能性。这和使用约束时得到的结果相同。

使用条件的构造一般足以覆盖所有非对称关系。可以看出，这种方法的好处是数值 0.3 和 0.1 在 Flip 元素中，您可以使它们取决于模型的其他方面。例如，假定您希望学习两个相邻元素颜色相同和颜色不同的可能性比率，可以创建一个元素，使用 Beta 分布表示这个权重。然后，您可以在 Flip 中用这个元素代替 0.3。这样，根据数据，可以学习到权重值上的一个分布。

5.1.3 直接和间接依赖性

在转向贝叶斯和马尔科夫网络之前，我希望介绍一个重点。典型的概率模型有许多对变量，对一个变量的认识会改变您对其他变量的观念。从定义上说，这些都是变量之间的依赖性。但是大部分依赖性是**间接**的：它们不会直接出现在两个变量之间，而是通过某些中介变量起作用。确切地讲，关于第一个变量的知识改变您对第二个变量的观念的原因是，关于第一个变量的知识改变了您对中介变量的观念，进而改变了对第二个变量的观念。

例如，图 5-2 中有 3 个变量：Printer Power Button On（打印机电源按钮开启）、Printer State（打印机状态）和 Number of Printed Pages（打印页面数量）。很明显，Printer Power Button On 和 Number of Printed Pages 有依赖性。如果您知道电源按钮关闭，将认为打印页面数为 0。但是在图中，从 Printer Power Button On 到 Number of Printed Pages 的依赖性经过了一个中介变量 Printer State。这意味着，关于 Printer Power Button On 的知识改变关于 Number of Printed Pages 的观念，其原因是它首先改变了您关于 Printer State 的观念，因而改变了关于 Number of Printed Pages 的观念。如果您知道电源按钮关闭，说明打印机停机，使您相信不会打印任何页面。

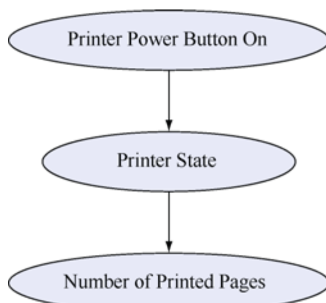


图 5-2　Printer Power Button On 和 Number of Printed Pages 通过中介变量产生间接关系

术语警告：之前我曾经谈到过有向和无向依赖性，现在谈论的则是直接和间接依赖性。尽管名称类似，但是它们的含义不同。直接依赖性直接出现在两个变量之间，其反义词是间接依赖性，通过中介变量体现。有向依赖性具有从一个变量到另一个变量的方向，而无向依赖性正相反，没有任何方向。您可能会遇到直接无向依赖性和间接有向依赖性。

我们再来看看另一个例子，这次涉及的是无向依赖性。之前，我曾经提供了一个图像中相邻像素之间无向依赖性的例子。不相邻的像素又是什么样的情况呢？考虑图 5-3 中的例子。如果您知道像素 11 是红色的，这将会导致您相信像素 12 可能是红色的，相应地让您相信像素 13 很可能是红色的。这是间接依赖性的明显例子，因为关于像素 11 的知识只有通过中介变量（像素 12）才能影响对像素 13 的观念。

重要的是认识您的领域中，哪些依赖性是直接的，哪些是间接的。在贝叶斯和马尔科夫网络中，您都创建一个图，图中变量之间的边表示依赖性。在贝叶斯网络中，这些边都是有向边，而在马尔科夫网络中则是无向边。您只对直接依赖性绘制边。如果两个变量只有间接依赖性，两者之间就没有边。

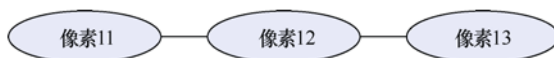


图 5-3　像素 11 和像素 13 通过中介变量像素 12 产生间接关系

接下来，您将研究贝叶斯网络和马尔科夫网络，前者是表示有向依赖性的模型，后者则是表示无向依赖性的模型。但是，我要指出，仅因为有向和无向依赖性各有不同的建模框架，并不意味着必须为您的模型选择其中一种框架。其他的框架在单一模型中组合了两类依赖性。在概率程序中这很容易做到：使用元素的生成定义编码有向依赖性，并添加用于表示无向依赖性的约束。

5.2　使用贝叶斯网络

您已经知道，编码变量之间的关系，对于概率建模是必不可少的。在本节中，您将学习用有向依赖性编码不对称关系的标准框架——贝叶斯网络。在第 4 章中的伦勃朗示例中已经看到了贝叶斯网络。本节提供更全面的介绍，包括您可以使用的推理模式的完整定义和解释。

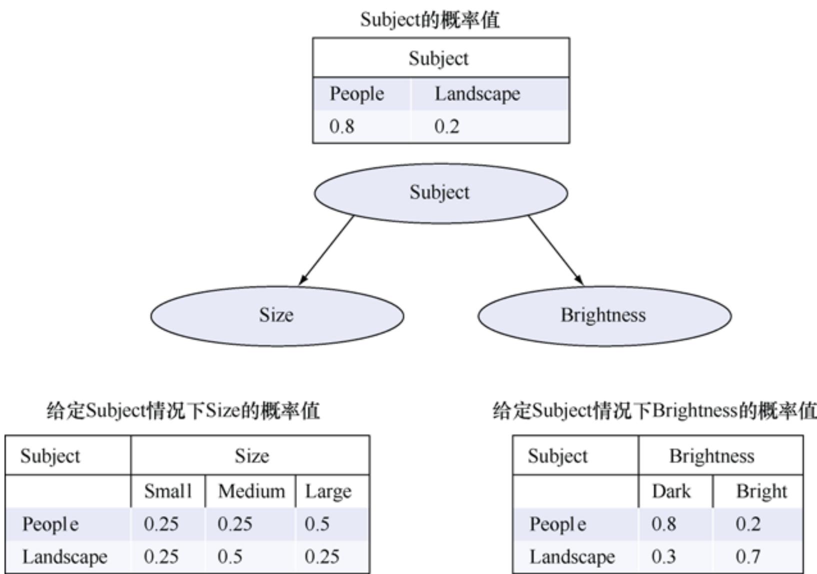
5.2.1　贝叶斯网络定义

贝叶斯网络是由 3 个部分组成的概率模型表现形式。

- 一组具有对应定义域的变量。
- 一个有向非循环图，每个变量是图的一个节点。
- 每个变量在其父变量给定情况下的条件概率分布（CPD）。

一组具有对应定义域的变量

图 5-4 中的例子显示 3 个变量：Subject、Size 和 Brightness。变量的定义域规定了该变量的可能值。Subject 的定义域是{People, Landscape}，Size 的定义域是{Small, Medium, Large}，Brightness 的定义域是{Dark, Bright}。


Subject的概率值

Subject	
People	Landscape
0.8	0.2

给定Subject情况下Size的概率值

Subject	Size		
	Small	Medium	Large
People	0.25	0.25	0.5
Landscape	0.25	0.5	0.25

给定Subject情况下Brightness的概率值

Subject	Brightness	
	Dark	Bright
People	0.8	0.2
Landscape	0.3	0.7

图 5-4　一个三节点贝叶斯网络

有向非循环图

有向意味着图中的每个边都有方向，它从一个变量指向另一个变量。第一个变量称为**父变量**，第二个变量称为**子变量**。在图 5-4 中，Subject 是 Size 和 Brightness 的父变量。**非循环**意味着图中没有环：没有沿箭头方向的有向环；您不能从一个节点开始，沿着箭头回到同一个节点。但是可以有**无向环**，即忽略方向的情况下可以有环。这一概念在图 5-5 中说明。左侧的图有一个有向环：A-B-D-C-A。在右侧的图中，环 A-B-D-C-A 有时候与箭头的方向相反，所以它是一个无向环。因此，左侧的图是不允许的，但是右侧的图是合法的。这一点很重要，因为以后当您使用无向边表达对称依赖性时，就可以使用无向环。

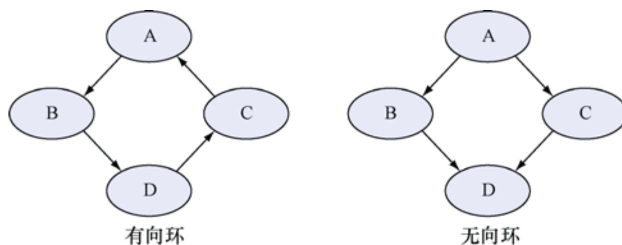


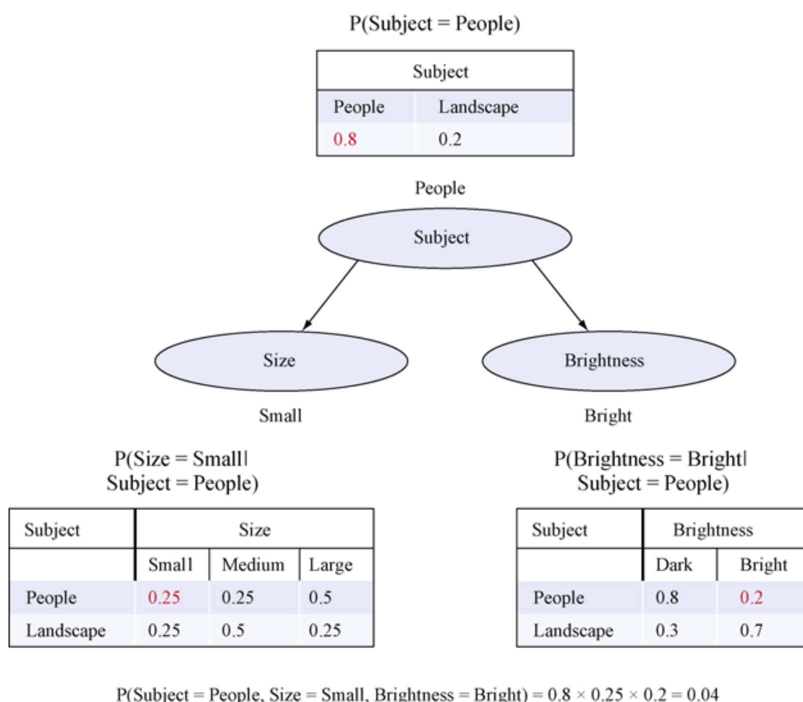
图 5-5　有向环是沿箭头方向并终止于起点的环

变量上的条件概率分布

CPD 描述在父变量值给定的情况下子变量的概率分布。这种 CPD 考虑了父变量的每个可能值，父变量可能是定义域中的任何值。对于每个赋值，它都定义了子变量上的一个概率分布。在图 5-6 中，每个变量都有一个 CPD。Subject 是网络的根，所以它没有父变量。当一个变量没有父变量时，CPD 指定变量上的单一概率分布。在本例中，Subject 取值 People 的概率为 0.8，取值 Landscape 的概率为 0.2。Size 的父变量为 Subject，所以 CPD 对每个 Subject 的值有一行。CPD 说明，当 Subject 取值为 People 时，Size 上的概率分布为：取值 Small 的概率为 0.25，Medium 的概率为 0.25，Large 的概率为 0.5。当 Subject 取值为 Landscape 时，Size 有不同的分布。最后，Brightness 的父变量也为 Subject，它的 CPD 对每个 Subject 值也有一行。

5.2.2　贝叶斯网络如何定义概率分布

贝叶斯网络的定义已经全部介绍完毕。下面，我们来看看贝叶斯网络是如何定义概率分布的。您需要做的第一件事是定义可能世界。对贝叶斯网络来说，可能世界由每个变量的赋值组成，确保每个变量都在其定义域内。例如，<Subject = People, Size = Small, Brightness = Bright>就是一个可能世界。



Subject	
People	Landscape
0.8	0.2

Subject	Size		
	Small	Medium	Large
People	0.25	0.25	0.5
Landscape	0.25	0.5	0.25

Subject	Brightness	
	Dark	Bright
People	0.8	0.2
Landscape	0.3	0.7

图 5-6　将每个 CPD 中的对应输入项相乘，计算可能世界的概率

接下来，定义可能世界的概率。这很简单。您所需要做的只是确定每个变量 CPD 中与可能世界中的父变量和子变量值匹配的条目。这一过程如图 5-6 所示。例如，对于可能世界<Subject = People, Size = Small, Brightness = Bright>，Subject 的输入项为 0.8，这可以从标记为 People 的列中找到。对于 Size，您可以查看对应于 Subject = People 的行和对应于 Size = Small 的列，得到输入项 0.25。最后，对于 Brightness，您仍然查看对应于 Subject = People 的行，这次取标记为 Bright 的列，获得输入项 0.2。将这些输入项相乘，得到可能世界的概率 $0.8 \times 0.25 \times 0.2 = 0.04$。

对每个可能世界执行这一过程，概率总和将为 1，这是应有的结果，对于贝叶斯网络来说总是如此。这样，您就看到了贝叶斯网络定义有效概率分布的方式。理解了贝叶斯网络的组成和含义之后，我们来看看如何使用这种网络，根据其他变量的知识得出对某些变量的观念。

5.2.3　用贝叶斯网络进行推理

贝叶斯网络编码变量之间的许多独立性。回忆一下，两个变量之间的独立性意味着学习到一个变量的相关知识，不能为您提供任何关于其他变量的新信息。从前面的例子

可以看到，Number of Printed Pages 和 Printer Power Button On **不是**独立的。当您知道没有打印任何页面时，电源按钮开启的概率就下降了。

条件独立性与此类似。如果**在第三个变量已知之后**，学习到第一个变量的相关知识不能为您提供第二个变量的任何相关信息，则两个变量在给定的第三变量下条件独立。**d-分离**标准决定了贝叶斯网络中的两个变量何时条件独立于第三组变量。这一标准有点复杂，所以我不提供正式的定义，而是描述基本原理并展示几个例子。

基本的思路是，推理**沿着**从一个变量到另一变量的**路径流动**。在图 5-4 的例子中，推理可能沿着 Size-Subject-Brightness 的路径，从 Size 流向 Brightness。在 5.1.3 小节中关于直接和间接依赖性的段落中您已经看到了这种思路。在间接依赖性中，推理通过中介变量从一个变量流向另一个变量。在这个例子中，Subject 是 Size 和 Brightness 之间的中介变量。在贝叶斯网络中，只要这条路径不会在某个变量上**阻塞**，推理就可以沿着路径前进。

在大部分情况下，如果变量需要观测，则路径被阻塞。所以，如果 Subject 需要观测，路径 Size-Subject-Brightness 被阻塞。这意味着，如果 Subject 也被观测，则观测 Size 不会改变对 Brightness 的观念。另一种说法是，Size 在 Subject 给定的情况下条件独立于 Brightness。在我们的模型中，画家选择的主题决定了画的尺寸和鲜艳度，但是在选择主题之后，尺寸和鲜艳度是独立生成的。

会聚箭头和诱导依赖性

另一种情况似乎有些违反直觉。在这种情况下，如果变量不能观测，则一条路径被阻塞，而在测得该变量时，路径畅通。我将扩展例子（如图 5-7 所示），以描述这种情况。您有一个新的变量 Material（材料），可能是油画颜料、水彩或者其他材料。Material 自然是 Brightness 的原因之一（油画可能比水彩画更鲜艳），所以网络中有一条从 Material 到 Brightness 的有向边。这里，同一个子变量有两个父变量，称为**会聚箭头**模式，因为从 Subject 和 Material 出发的边会聚于 Brightness。

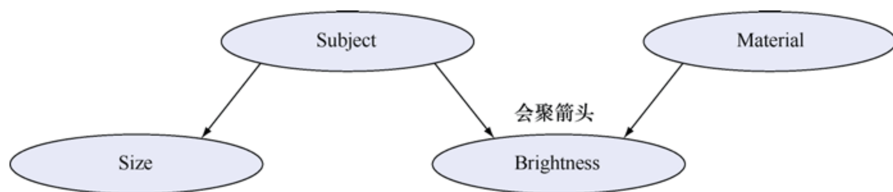


图 5-7　扩展的绘画示例，包括同一子变量的两个父变量之间的会聚箭头

现在，我们考虑一下 Subject 和 Material 之间的推理。根据模型，Subject 和 Material 是独立生成的。所以以下事实成立。

（1）在没有观察到任何证据时，Subject 和 Material 是独立的。

但是，当您观察到画作是鲜艳的时，会发生什么呢？根据我们的模型，风景画通常比人物画更鲜艳。在观察到画作是鲜艳的之后，您将推理，这幅画更有可能是风景画。

假定之后您观察到这幅画是油画，这种画也更可能有鲜艳的色彩。这个观测值提供了画作鲜艳度的另一个解释。因此，这幅画是风景画的概率和观察到画作鲜艳之后、观察到该画是油画之前相比有所降低。您可以看到，推理沿着 Material-Brightness-Subject 的路径，从 Material 流向 Subject。所以，您得到了如下的陈述：

（2）在 Brightness 给定的情况下，Subject 和 Material 不是条件独立的。

您现在所拥有的模式与通常的模式相反。有一条路径在中介变量不能测得时阻塞，在观测到该变量时畅通。这种情况称作**诱导依赖性**——两个变量之间的依赖性是对第三个变量的观测所诱导的。贝叶斯网络中任何会聚箭头模式都可能导致诱导依赖性。

两个变量之间的路径可能既包含常规模式，又包含会聚箭头。沿该路径流动的推理只有在路径上的任何节点都没有被阻塞时才能进行。图 5-8 展示了 Size-Subject-Brightness-Material 路径的 4 个例子。在左上角的例子中，Subject 和 Brightness 都没有观测到，该路径在 Brightness 上被阻塞，因为它有会聚箭头。在右侧的下一个例子中观测到了 Subject，所以路径阻塞在 Subject 和 Brightness 上。在左侧的下一个例子中，没有观测到 Subject 而观测到了 Brightness，这是路径在 Subject 和 Brightness 上都没有阻塞的必要条件。最后，右下角的例子中除了 Subject 之外，还观测到了 Brightness，所以路径在此处阻塞。

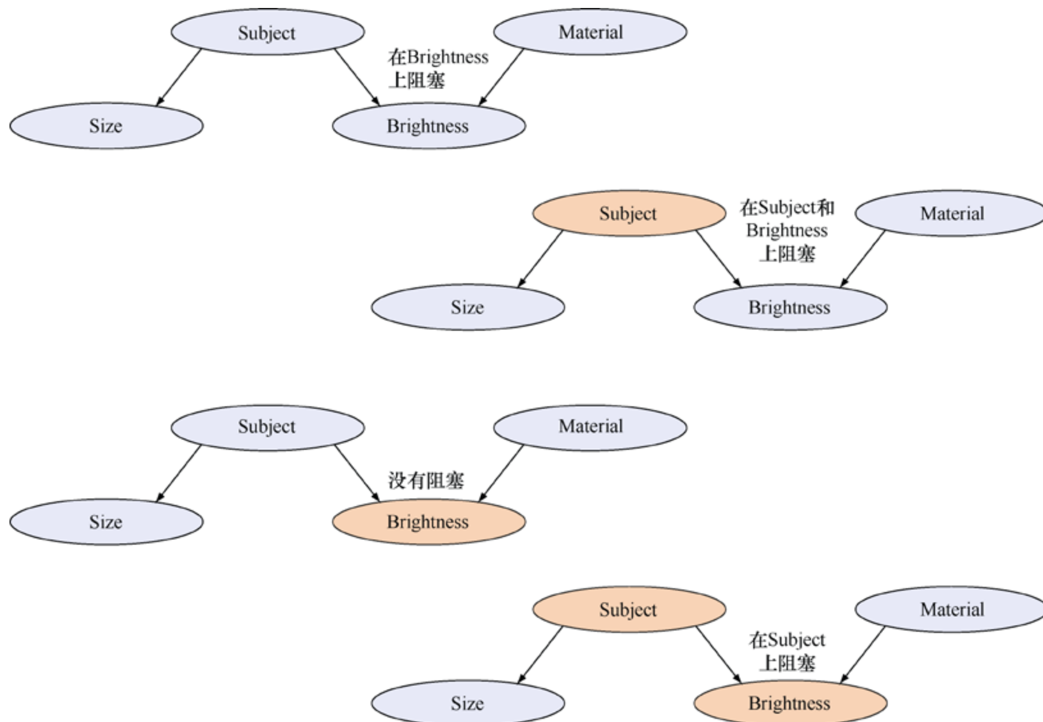


图 5-8　阻塞和畅通路径的例子，结合了常规模式和会聚箭头模式。每张图展示了在观测或者未观测 Subject 和 Brightness 的情况下，从 Size 到 Material 的路径

5.3 探索贝叶斯网络的一个示例

您已经学习了贝叶斯网络的基本概念，让我们来研究一个打印机故障诊断的示例。我首先展示网络的设计方法，然后展示用网络进行推理的所有途径。我将把网络参数学习的讨论留到第 12 章，在那一章中您将探索参数学习的一种实用设计模式。

5.3.1 设计一个计算机系统诊断模型

想象一下，您打算设计一个用于技术支持的服务台应用，希望帮助技术支持人员尽快识别故障原因。您可以为这个应用程序使用一个概率推理系统——根据由用户和诊断测试报告中提供的证据，确定系统的内部状态。对于这样的应用，使用贝叶斯网络表示概率模型是很自然的。

在设计贝叶斯网络时，通常采取三个步骤：选择变量及对应的定义域、说明网络结构和编码 CPD。您将看到在 Figaro 中如何完成这三个步骤。

在实践中，您通常不以线性方式完成所有步骤：选择所有变量，构建整个网络结构，然后写下 CPD。通常，您将一次构建网络的一小部分，并逐步求精。在此我们采用的就是这种方法。首先，为一般打印机故障模型构建一个网络，然后深入更详细的打印机模型。

一般打印机故障模型：变量

您希望建立用户报告、可能涉及的故障以及导致这些故障的系统因素的模型，并将为此引入变量。

首先是 Print Result Summary（打印结果摘要），这个变量以高度抽象的方式表示用户对打印结果的总体体验。当用户第一次致电服务台时，他可能只提供这样的摘要。可能结果有三：（1）打印很完美（您将其标记为“出色”）；（2）能打印，但是效果不是太好（“差”）；（3）完全不能打印（“无”）。

接下来考虑打印结果的各个具体方面，包括：Number of Printed Pages（打印页数），可能为 0、其中一些页面或者所有页面；Prints Quickly（是否快速打印）是一个布尔变量，表示打印是否在合理时间内完成；Good Print Quality（高质量打印）也是布尔变量。单独建立各个方面模型的原因之一是这样可以区分不同的故障。例如，如果没有打印任何页面，可能是因为网络中断。但是如果打印了一部分页面，问题就不太可能出在网络上，而更可能是用户的错误。

接着，考虑可能影响打印结果的所有系统要素，包括：Printer State（打印机状态），可能为好、差或者停机；Software State（软件状态），可能为正确、有小故障或者崩溃；Network State（网络状态），可能为正常、断续或者中断；User Command Correct（用户

命令正确）是一个布尔变量。

一般打印机故障模型：网络结构

您已经定义的变量有三组：抽象的打印结果摘要、打印结果的各个具体方面以及影响打印结果的系统状态。相应地，将网络设计为 3 个层次，是有意义的做法。这些层次应该采用什么顺序？

系统状态变量和具体的打印结果变量之间存在因果关系。例如，Network State（网络状态）为中断是无法快速打印（Print Quickly）的原因。此外，单独打印结果变量和整体打印结果摘要（Print Result Summary）之间存在具体-抽象关系。在 5.1.1 小节中我曾经讲过，这些关系可能有两个方向。在我们的应用中，您所要建立的是用户体验和打印结果报告的模型，所以根据我介绍的经验法则，正确的方向是从具体打印结果变量指向抽象摘要。所以，我们的网络层次顺序为：（1）系统状态变量，（2）具体打印结果变量 （3）打印结果摘要。

网络结构如图 5-9 所示。您可以看到 3 个层次，但是一个层次中的每个变量并不都有指向下一层次中变量的边。这是因为某些打印结果变量只依赖于一些系统组件的状态。例如，打印质量好坏取决于打印机的状态，但是不取决于网络。同样，根据我们的模型，打印速度只取决于网络和软件。这些陈述是否正确有待争论；我们所要指出的要点是，在任何应用中，可以用参数删除某些边。删除一些边的好处是可以得到较小的 CPD。

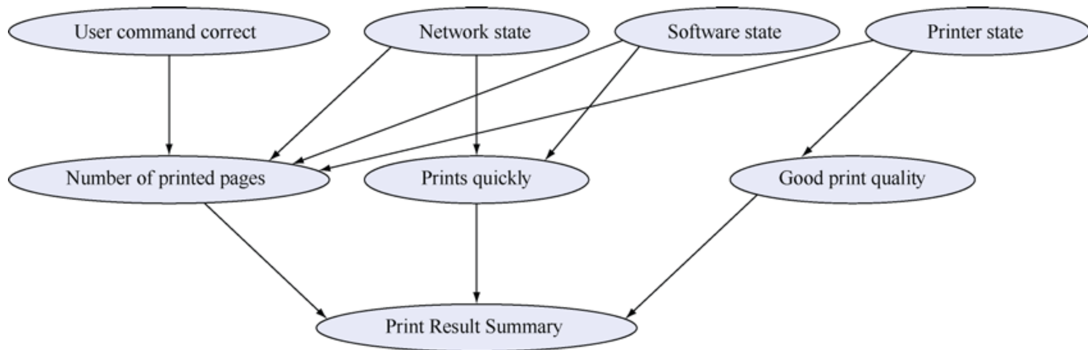


图 5-9 我们的计算机系统诊断模型中一般打印故障部分的网络结构

一般打印机故障模型：CPD

我将通过 Figaro 代码向您展示 CPD 的设计，确保解释其中最有趣的项目。5.1.1 小节说明了用 Figaro 定义 CPD 的各种方法。在此您将使用其中的几种。

程序清单 5-1　用 Figaro 实现一般打印机故障模型

```
val printerState = …

val softwareState =
  Select(0.8 -> 'correct, 0.15 -> 'glitchy, 0.05 -> 'crashed)
val networkState =
  Select(0.7 -> 'up, 0.2 -> 'intermittent, 0.1 -> 'down)
val userCommandCorrect =
  Flip(0.65)

val numPrintedPages =
  RichCPD(userCommandCorrect, networkState,
          softwareState, printerState,
           (*, *, *, OneOf('out)) -> Constant('zero),
           (*, *, OneOf('crashed), *) -> Constant('zero),
           (*, OneOf('down), *, *) -> Constant('zero),
           (OneOf(false), *, *, *) ->
             Select(0.3 -> 'zero, 0.6 -> 'some, 0.1 -> 'all),
           (OneOf(true), *, *, *) ->
             Select(0.01 -> 'zero, 0.01 -> 'some, 0.98 -> 'all))

val printsQuickly =
  Chain(networkState, softwareState,
        (network: Symbol, software: Symbol) =>
           if (network == 'down || software == 'crashed)
                  Constant(false)
           else if (network == 'intermittent || software == 'glitchy)
                  Flip(0.5)
           else Flip(0.9))

val goodPrintQuality =
  CPD(printerState,
     'good -> Flip(0.95),
     'poor -> Flip(0.3),
     'out -> Constant(false))

val printResultSummary =
  Apply(numPrintedPages, printsQuickly, goodPrintQuality,
        (pages: Symbol, quickly: Boolean, quality: Boolean) =>
          if (pages == 'zero) 'none
          else if (pages == 'some || !quickly || !quality) 'poor
          else 'excellent)
```

这来自于下面介绍的详细打印机模型

对于根变量，我们采用 Select 或者 Flip 等原子 CPD

numPrintedPages 用 RichCPD 表示用户命令、网络、软件和打印机状态上的依赖性

printsQuickly 使用 Chain 表示网络和软件状态上的依赖性

goodPrintQuality 使用简单的 CPD 表示打印机状态上的依赖性

printResultSummary 完全由其父变量决定，所以使用 Apply

上述代码使用了多种技术以表现子变量与其父变量的概率依赖。

- numPrintedPages 用 RichCPD 实现如下逻辑：如果打印机状态为停机（out）或者网络状态为中断（down）或者软件崩溃（crashed），不管其他父变量状态为何，打印页数均为 0。否则，如果用户发出错误的命令，就不太可能打印所有页面，但是如果发出的是正确的命令，就很有可能打印所有页面。
- printsQuickly 用 Chain 实现如下逻辑：如果网络中断或者软件崩溃，打印机的打印速度肯定不快。否则，如果网络断断续续或者软件有小问题，能否快速打印就不确定。如果网络和软件都处于良好状态，打印通常很快（但是不能保证）。
- goodPrintQuality 使用简单的 CPD。如果打印机停机，肯定不会有高质量的打印。如果打印机处于很差的状态，可能也不会有高质量的打印。即使打印机处于良好状态，也不能保证高质量的打印（因为打印机就是这样的）。
- printResultSummary 是一个确定的变量：它完全由其父变量决定，没有任何不确定性。您可以使用 Apply 代替 Chain 表示确定的变量。

详细的打印机模型

本节将较快地介绍详细的打印机模型，因为许多原理都是相同的。网络结构如图 5-10 所示。打印机状态受到 3 个因素的影响：Paper Flow（纸张流动）、Toner Level（墨粉水平）和 Printer Power Button On（打印机电源按钮开启）。该模型增加了一类前所未见的新变量——指标或者计量值。Paper Jam Indicator On（卡纸指示灯亮）是 Paper Flow 的计量值，Toner Low Indicator On（墨粉不足指示灯亮）是 Toner Level 的计量值。正如 5.1.1 小节中所介绍的，真值与其计量值之间的关系是一种因果关系，所以模型中有从 Paper Flow 到 Paper Jam Indicator On 和从 Toner Level 到 Toner Low Indicator On 的边。

下面是定义 CPD 的代码，基本上都很简单。

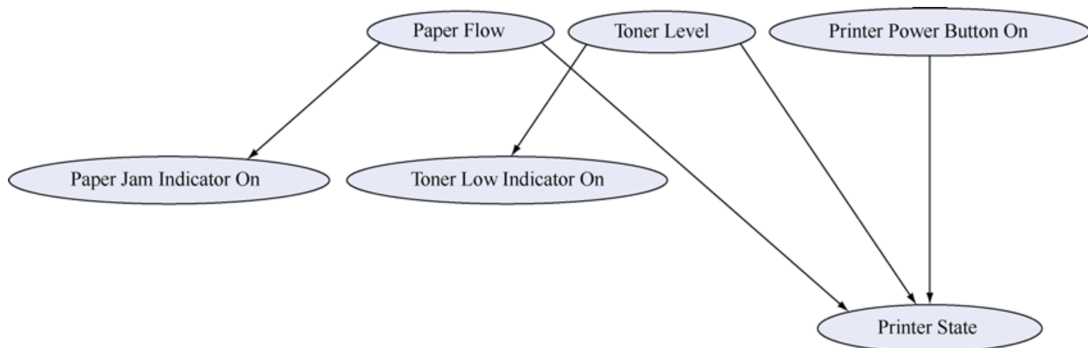


图 5-10　详细打印机模型的网络结构

程序清单 5-2 Figaro 中的详细打印机模型

```
val printerPowerButtonOn = Flip(0.95)
val tonerLevel = Select(0.7 -> 'high, 0.2 -> 'low, 0.1 -> 'out)
val tonerLowIndicatorOn =
  If(printerPowerButtonOn,
    CPD(tonerLevel,
        'high -> Flip(0.2),
        'low -> Flip(0.6),
        'out -> Flip(0.99)),
    Constant(false))
val paperFlow = Select(0.6 -> 'smooth, 0.2 -> 'uneven, 0.2 -> 'jammed)

val paperJamIndicatorOn =
  If(printerPowerButtonOn,
     CPD(paperFlow,
         'smooth -> Flip(0.1),
         'uneven -> Flip(0.3),
         'jammed -> Flip(0.99)),
    Constant(false))

val printerState =
  Apply(printerPowerButtonOn, tonerLevel, paperFlow,
        (power: Boolean, toner: Symbol, paper: Symbol) => {
          if (power) {
            if (toner == 'high && paper == 'smooth) 'good
            else if (toner == 'out || paper == ' jammed) 'out
            else 'poor
          } else 'out
        })
```

回忆一下，单引号'表示 Scala 符号类型

If 中嵌套了一个 CPD。如果打印机电源按钮开启，墨粉不足指示灯取决于墨粉水平。如果电源按钮关闭，不管墨粉水平为何，墨粉指示灯将关闭，因为没有电源

打印机状态是组成因素的确定性摘要

总结这个例子，完整的贝叶斯网络结构如图 5-11 所示。您也可以在本书代码中的 chap05/PrinterProblem.scala 下看到完整的程序。

5.3.2 用计算机系统诊断模型进行推理

本节说明如何用刚刚构建的计算机系统诊断模型进行推理。Figaro 的推理机制很简单，但是从中得到的推理模式很有趣，说明了 5.2.3 小节中介绍的概念。本节的所有推理模式划分为本章代码中的单独步骤，可以为不希望执行的步骤加上注释，凸显介绍的每个步骤。

查询先验概率

首先，我们查询没有任何证据时打印机电源按钮开启的概率，这是一个先验概率。可以使用如下代码：

```
val answerWithNoEvidence =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Prior probability the printer power button is on = " +
  answerWithNoEvidence)
```

计算 P(Printer Power Button On = true)

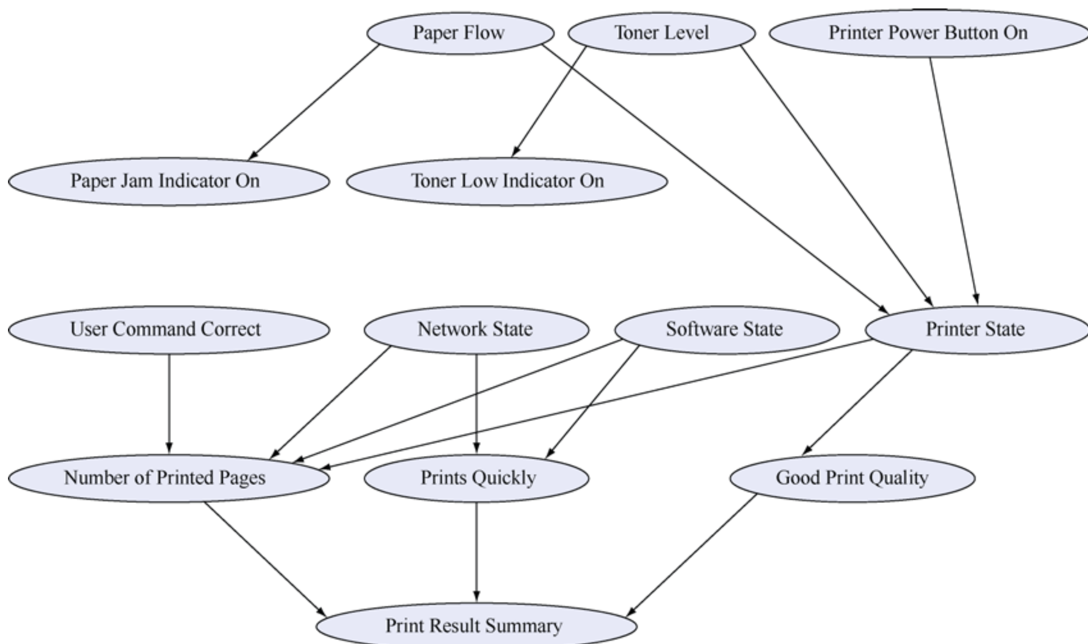


图 5-11　计算机诊断示例的完整网络

这段代码打印如下结果：

```
Prior probability the printer power button is on = 0.95
```

回顾模型，就将发现 printerPowerButtonOn 用如下代码行定义：

```
val printerPowerButtonOn = Flip(0.95)
```

可以看到，如果忽略除了这个定义之外的整个模型，您所得到的就是查询的答案。这是如下通用规则的一个例子：网络中某个变量的下游只在有证据的时候才与该变量相关。特别是，对于没有任何证据的先验概率，您不关心网络的下游部分。

在有证据的情况下查询

如果您引入了证据，会发生什么情况？我们在已知打印结果不佳（暗示着有一些结果，但是不是用户所要的结果）的情况下，查询打印机电源按钮开启的概率。可以使用如下代码：

```
printResultSummary.observe('poor)
val answerIfPrintResultPoor =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Probability the printer power button is on given a poor "
  + "result = " + answerIfPrintResultPoor)
```

计算 P(Printer Power Button On = true | Print Result = poor)

这段代码打印如下结果：

```
Probability the printer power button is on given a poor result = 1.0
```

这可能有些令人惊讶！概率高于您没有任何关于打印的证据时。考虑一下模型，就能知道其中的原因。糟糕的打印结果只有在至少打印出某些页面的情况下才能发生，这不是电源关闭的结果。因此，电源开启的概率为 1。

现在用“什么都没有打印”的证据进行查询：

```
printResultSummary.observe('none)
val answerIfPrintResultNone =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Probability the printer power button is on given empty "
  + "result = " + answerIfPrintResultNone)
```

计算 P(Printer Power Button On = true | Printer Result = none)

结果如下：

```
Probability the printer power button is on given empty result =
0.8573402523786461
```

这符合您的预期。电源按钮关闭很好地解释了没有打印结果的情况，所以根据证据，它的概率增大了。

独立与阻塞

5.2.3 小节介绍了阻塞路径的概念以及这个概念与条件独立性的关系。这个概念可以用 3 个变量说明：Print Result Summary、Printer Power Button On 和 Printer State。在图 5-11 中，可以看到从 Printer Power Button On 到 Print Result Summary 的路径通过了 Printer State。因为这不是会聚箭头模式，该路径在观测到 Printer State 时被阻塞。确实，现在可以看到这种情况：

```
printResultSummary.unobserve()
printerState.observe('out)
val answerIfPrinterStateOut =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Probability the printer power button is on given " +
        "out printer state = " + answerIfPrinterStateOut)
```

计算 P(Printer Power Button On = true | Printer State = out)

```
printResultSummary.observe('none)
val answerIfPrinterStateOutAndResultNone =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Probability the printer power button is on given " +
        "out printer state and empty result = " +
        answerIfPrinterStateOutAndResultNone)
```

计算 P(Printer Power Button On = true | Printer State = out, Print Result Summary = none)

上述代码打印如下结果：

```
Probability the printer power button is on given out printer state =
0.6551724137931032
Probability the printer power button is on given out printer state and empty
result = 0.6551724137931033
```

可以看到，在已知打印机状态为停机之后，得知打印结果为空并没有改变打印机电源按钮开启的概率。这是因为，在 Printer State 给定的情况下，Print Result Summary 条件独立于 Printer Power Button On。

在相同原因的不同结果之间推理

目前为止看到的所有推理路径是沿着网络向上的。在推理时您也可以组合两个方向。当您考虑计量值所提供的情况，以及依照这些情况得出的结论，就很容易理解这一点。在我们的例子中，Toner Low Indicator On 是 Toner Level 的子变量，Toner Level 是 Printer State 的父变量。如果墨粉水平很低，打印机状态就不太可能好。同时，墨粉不足指示灯亮是墨粉水平低的信号。如果观察到墨粉不足指示灯亮，打印机状态良好的概率降低是合理的。您可以在如下代码中看到这种情况：

```
printResultSummary.unobserve()
printerState.unobserve()
val printerStateGoodPrior =
  VariableElimination.probability(printerState, 'good)
println("Prior probability the printer state is good = "
        + printerStateGoodPrior)
```

计算 P(Printer State = good)

```
tonerLowIndicatorOn.observe(true)
val printerStateGoodGivenTonerLowIndicatorOn =
  VariableElimination.probability(printerState, 'good)
println("Probability printer state is good given low toner "
        + "indicator = " + printerStateGoodGivenTonerLowIndicatorOn)
```

计算 P(Printer State = good | Toner Low Indicator On = true)

这段代码打印如下结果：

```
Prior probability the printer state is good = 0.39899999999999997
Probability the printer state is good given low toner indicator =
0.23398328690807796
```

可以看到，在您观察到墨粉不足指示灯时，打印机状态良好的概率下降，这正如预期。

在同一结果的不同原因之间推理：诱导依赖性

正如 5.2.3 小节中所讨论的，在同一结果的不同原因之间推理与其他类型的推理不同，因为它涉及会聚箭头，这会导致诱导依赖性。我们以 Software State 和 Network State 为例，它们都是 Prints Quickly 的父变量。首先，您计算软件状态正确的概率：

```
tonerLowIndicatorOn.unobserve()
val softwareStateCorrectPrior =
  VariableElimination.probability(softwareState, 'correct)
println("Prior probability the software state is correct = " +
       softwareStateCorrectPrior)
```

计算 P(Software State = correct)

打印结果如下：

```
Prior probability the software state is correct = 0.8
```

接下来，您将观察到网络正常连接，并再次查询软件状态：

```
networkState.observe('up)
val softwareStateCorrectGivenNetworkUp =
  VariableElimination.probability(softwareState, 'correct)
println("Probability software state is correct given network up = " +
        softwareStateCorrectGivenNetworkUp)
```

计算 P(Software State = correct | Network State = up)

打印结果如下：

```
Probability software state is correct given network up = 0.8
```

尽管从 Network State 通过 Prints Quickly 到 Software State 有一条清晰的路径，但是概率没有变化！这说明，一般来说同一个结果的两个原因是相互独立的。从直觉上这很正确：网络正常运行对软件状态是否正确毫无影响。

现在，如果您知道打印机速度不快，情况就不一样了。如果打印机速度较慢，我们的模型提供了两种可能的解释：网络问题或者软件问题。如果观察到网络正常连接，则必然是软件问题。您可以用如下代码看到这种情况：

计算 P(Software State = correct | Prints Quickly = false)

```
networkState.unobserve()
printsQuickly.observe(false)
val softwareStateCorrectGivenPrintsSlowly =
  VariableElimination.probability(softwareState, 'correct)
println("Probability software state is correct given prints "
        + "slowly = " + softwareStateCorrectGivenPrintsSlowly)
```

计算P(Software State = correct | Prints Quickly = false, Network State = up)

```
networkState.observe('up)
val softwareStateCorrectGivenPrintsSlowlyAndNetworkUp =
  VariableElimination.probability(softwareState, 'correct)
println("Probability software state is correct given prints "
        + "slowly and network up = "
        + softwareStateCorrectGivenPrintsSlowlyAndNetworkUp)
```

运行上述代码打印如下结果：

```
Probability software state is correct given prints slowly =
0.6197991391678623
Probability software state is correct given prints slowly and network up =
 0.39024390243902435
```

得知网络正常连接，将显著地降低软件状态正确的概率。所以，Software State 和 Network State 是独立的，但是在 Prints Quickly 给定的情况下不是条件独立的，这是诱导依赖性的一个例子。

总结如下。

- 从结果 *X* 向间接原因 *Y* 推理时，*X* 不独立于 *Y*，但是如果 *Z* 阻塞从 *X* 到 *Y* 的路径，那么在 *Z* 给定的情况下，*X* 和 *Y* 条件独立。
- 从原因向间接结果推理或者在同一原因的两个结果之间推理时，上一点也成立。
- 对于同一结果 *Z* 的两个原因 *X* 和 *Y*，情况相反。*X* 与 *Y* 独立，但是在 *Z* 给定的情况下，由于诱导依赖性，两者不是条件独立的。

计算机系统诊断示例到此就圆满完成了。您已经看到了包含有趣推理模式、内容充实的网络。下一节，您将超越传统的贝叶斯网络。

5.4 使用概率编程扩展贝叶斯网络：预测产品的成功

本节说明如何扩展基本贝叶斯网络模型，以预测市场活动和产品布局的成功。这一例子的目的有两方面：首先，说明使用编程语言扩展贝叶斯网络的能力，其次，说明使用贝叶斯网络不仅能够推理观察到的事件起因，还能够预测未来的事件。和计算机系统诊断示例一样，我将首先展示模型的设计方法，并用 Figaro 表达，然后说明如何用这个模型推理。

5.4.1 设计产品成功预测模型

想象您有一个新产品，希望尽可能使其取得成功。实现这一目标有多种途径。您可以投资于产品的包装和其他吸引客户的方面；可以尝试以客户更容易接受的方式定价。或者，可以提供免费版本进行促销，希望人们与其朋友分享。在选择策略之前，您希望知道每个因素的相对重要性。

本节描述用于这个应用的模型框架。我称之为**框架**是因为它只是真正构建的模型的一个骨架，但是足以说明我的论点。本节提供的模型是一个只有 4 个节点的贝叶斯网络，但是节点和 CPD 的类型丰富且有趣。

该模型的 4 个变量如图 5-12 所示。

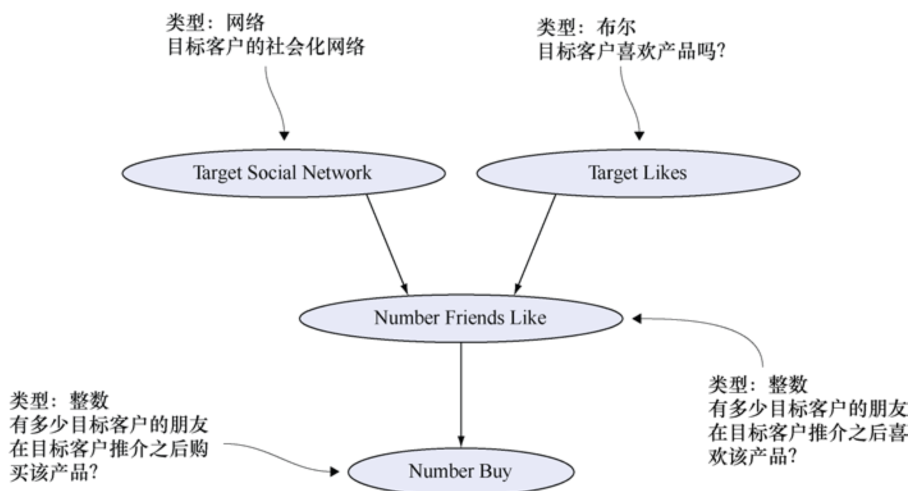


图 5-12　产品成功预测网络

- Target Social Network（目标社会化网络）变量的类型是一个社会化网络。这是编程语言让您超越常规贝叶斯网络的一种方式，在贝叶斯网络中变量只能是布尔型、枚举型、整数型或者实数型。这个变量的 CPD 随机地根据目标的流行程度生成网络。流行程度本身是一个控制变量，因此模型中将其作为已知的常量而非变量。但是您可以引入有关流行程度的不确定性，使其成为变量。
- Target Likes 是一个布尔变量，表示目标是否喜欢产品，这是产品质量的一个函数。同样，产品质量是一个已知的常量，但是您也可以使之成为取决于投资的变量。
- Number Friends Like 是一个整数变量，但是其 CPD 遍历 Target Social Network 以决定看到产品的朋友和喜欢该产品的朋友数量。这比贝叶斯网络中的传统 CPD 更加丰富。
- Number Buy 是一个整数变量，它的模型通过考虑每个喜欢产品的人购买产品的概率定义，这一概率取决于可承受性——这个控制变量的值是已知常数。因此，购买产品的人数呈二项分布。

下面是这个模型的 Figaro 代码。我将概要介绍代码的工作方式，然后介绍模型的细节。

程序清单 5-3　Figaro 中的产品成功预测模型

```
class Network(popularity: Double) {
  val numNodes = Poisson(popularity)
}

class Model(targetPopularity: Double, productQuality: Double,
            affordability: Double) {

  def generateLikes(numFriends: Int,
                    productQuality: Double): Element[Int] = {

    def helper(friendsVisited: Int, totalLikes: Int,
               unprocessedLikes: Int): Element[Int] = {
      if (unprocessedLikes == 0) Constant(totalLikes)
      else {
        val unvisitedFraction =
          1.0 – (friendsVisited.toDouble – 1) / (numFriends – 1)
        val newlyVisited = Binomial(2, unvisitedFraction)
        val newlyLikes =
          Binomial(newlyVisited, Constant(productQuality))
        Chain(newlyVisited, newlyLikes,
              (visited: Int, likes: Int) =>
               helper(friendsVisited + unvisited,
                      totalLikes + likes,
                      unprocessedLikes + likes - 1))
      }
    }
    helper(1, 1, 1)
  }

  val targetSocialNetwork = new Network(targetPopularity)

  val targetLikes = Flip(productQuality)

  val numberFriendsLike =
    Chain(targetLikes, targetSocialNetwork.numNodes,
          (l: Boolean, n: Int) =>
            if (l) generateLikes(n, productQuality)
            else Constant(0))

  val numberBuy =
    Binomial(numberFriendsLike, Constant(affordability))
}
```

定义一个 Network 类，该类包含一个由 Poisson 元素（参见正文）定义的属性

创建一个 Model 类，以已知的控制参数作为参数

定义生成喜欢产品的人数的递归过程（参见正文）

根据目标的流行程度，将目标社会化网络定义为一个随机网络

目标是否喜欢产品是基于产品质量的布尔元素

如果目标喜欢产品，用 generateLikes 计算朋友数量。如果目标不喜欢产品，就不会告诉朋友有关产品的事情，所以数量为 0

购买产品的朋友数呈二项分布（参见正文）

代码中有 3 个细节需要另外解释：Network 类使用的 Poisson 元素、generateLikes

过程和 numberBuy 定义。我们首先介绍 Poisson 元素和 numberBuy 逻辑，然后介绍模型中最有趣的部分——generateLikes 过程。

- Poisson **元素**是一个整数元素，使用所谓的**泊松分布**。泊松分布通常用于建立某一事件在一段时间内出现次数的模型，比如一个月内网络故障的次数或者一场足球赛中的角球次数。利用少许创意，在任何您想要知道某个范围内的事物数量的情况下，都可以用泊松分布作为模型。这里，您用这种分布建立某人社会化网络中的人数，这不同于通常的用法，但是仍是一个合理的选择。

  Poisson 元素有一个参数——在一段时间内预期出现的平均次数，但是允许该数值高于或者低于平均数。在这个模型中，参数是目标的流行程度；流行程度应该是您预计在目标客户社会化网络中的平均人数。
- 代码中有确定购买产品人数的逻辑。每个喜欢产品的人购买产品的概率等于可承受性参数的值。所以购买产品的人数由一个二项分布给出，其中的试验次数是喜欢产品的人数，购买概率取决于产品的可承受性。因为喜欢产品的人数本身是一个元素，必须使用复合二项分布，以该元素作为参数。复合二项分布元素要求试验成功概率也是一个元素，这就是将可承受性封装在一个常量（Constant）元素中的原因。常量元素取得一个常规 Scala 值，生成始终取该值的 Figaro 元素。
- generateLikes 函数的目的是向社会化网络包含给定人数的目标客户提供产品之后，喜欢产品的人数。这个函数假定目标本身喜欢该产品，否则，将不会调用该函数。它描述一个人们向朋友推荐自己喜欢的产品的随机过程。generateLikes 函数有两个参数：（1）目标客户社会化网络中的人数，这是一个整数；（2）产品质量，是 0～1 的双精度数。

  generateLikes 函数的逻辑并不关键，因为重点是可以使用有趣的递归函数作为 CPD。但是我将解释这一逻辑，使您能够看到一个示例。generateLikes 的大部分工作由一个助手函数完成。这个函数跟踪 3 个值。

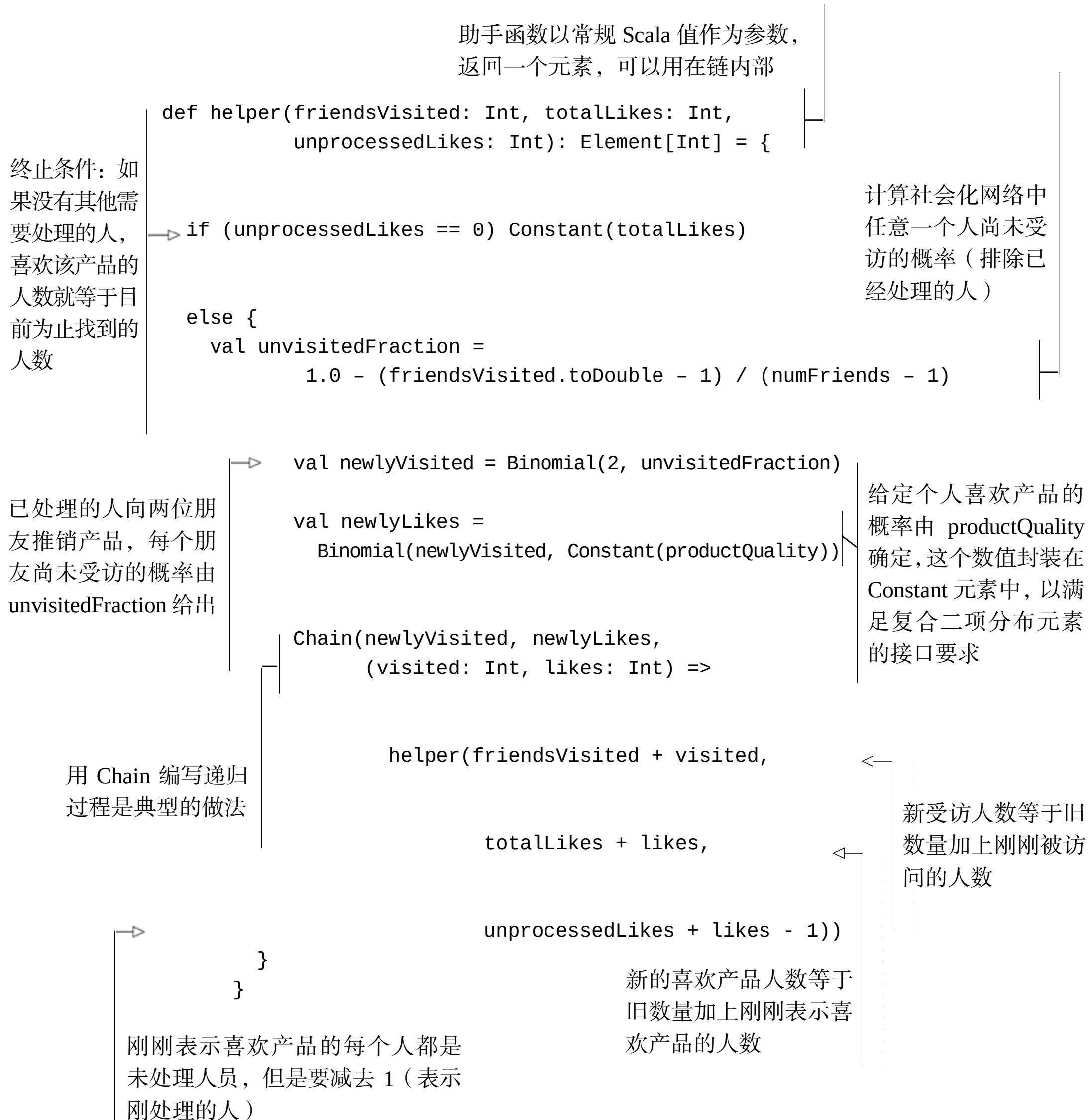
  —— friendsVisited 是目标社会化网络中已经得到产品信息的人数，最初为 1，因为开始只有目标客户已经了解产品信息。

  —— totalLikes 代表目前为止受访者中喜欢产品的人数。最初也为 1，因为假定目标客户喜欢产品，这是调用 generateLikes 的前提。

  —— unprocessedLikes 表示在您尚未模拟向其朋友推销的人中喜欢该产品的人数。

我将通过如下代码解释助手函数的逻辑。

程序清单 5-4 遍历社会化网络的助手函数



```
def helper(friendsVisited: Int, totalLikes: Int,
           unprocessedLikes: Int): Element[Int] = {
  if (unprocessedLikes == 0) Constant(totalLikes)
  else {
    val unvisitedFraction =
          1.0 – (friendsVisited.toDouble – 1) / (numFriends – 1)
    val newlyVisited = Binomial(2, unvisitedFraction)
    val newlyLikes =
      Binomial(newlyVisited, Constant(productQuality))
    Chain(newlyVisited, newlyLikes,
          (visited: Int, likes: Int) =>
            helper(friendsVisited + visited,
                   totalLikes + likes,
                   unprocessedLikes + likes - 1))
  }
}
```


这个例子使用了较多的编程和 Scala 技巧，但是主要的建模技术与贝叶斯网络类似，要点是想象一个生成可能世界的过程。在这个例子中，您看到产品通过社会化网络传播的一个相对简单的过程，但是很容易编写更丰富的过程。

5.4.2　用产品成功预测模型进行推理

因为您已经设计了用于预测成功的模型，典型的用法是设置控制变量值，预测购买产品的人数。因为人数是一个范围很广的整数变量，因此您对预测特定值的概率没有兴趣，而是想要知道预期的平均值。这称为该值的**期望**。

在概率论中，期望是一个通用概念。期望取得在变量上定义的一个函数，返回该函数的平均值。图 5-13 展示了一个例子，从任何类型的值上的概率分布开始；在这个例子中，这些值是整数。然后，对每个值应用一个函数产生一个双精度值。在这个例子中，函数将每个整数转换为其双精度表现形式。接着，取得双精度值的加权平均数，每个值的权重由其概率决定。这意味着，将每个双精度值乘以其概率并加总。

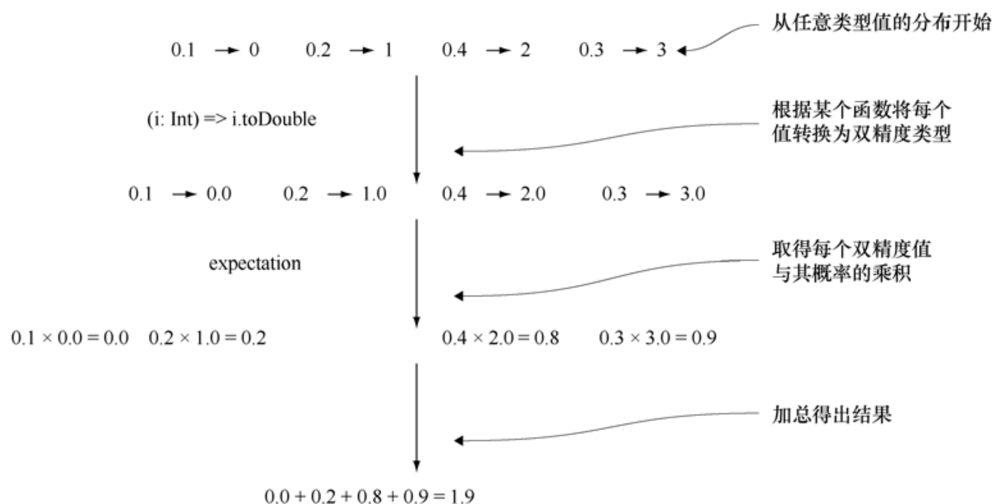


图 5-13　计算整数元素上的一个分布的期望值。元素的值首先转换为双精度值，然后取这些双精度值的加权平均值，权重为其概率

在预测产品成功的例子中，您想要计算购买产品人数（一个整数值）的期望值。可以在 Figaro 中使用如下代码行：

```
algorithm.expectation(model.numberBuy, (i: Int) => i.toDouble)
```

在此，algorithm 是您所使用的推理算法的一个句柄。本应用使用重要性抽样算法，这种算法特别适合于预测复杂生成过程的结果。因为您需要该算法的一个句柄，运行推理所需的代码比以前稍微复杂一些，下面的代码片段对此做出了解释。取得控制常量并计算购买产品期望值的整个过程由 predict 函数实现：

```
def predict(targetPopularity: Double, productQuality: Double,
        affordability: Double): Double = {

  val model =
    new Model(targetPopularity, productQuality, affordability)

  val algorithm = Importance(1000, model.numberBuy)

  algorithm.start()

  val result =
    algorithm.expectation(model.numberBuy, (i: Int) => i.toDouble)

  algorithm.kill()

  result
}
```

使用给定的控制常量，创建产品预测模型的新实例

创建重要性抽样算法的一个实例。1000 是样本数，model.numberBuy 表示我们想要预测的目标

运行算法

计算购买产品人数的期望值

清理并释放算法占用的资源

如果您试图理解购买产品人数上各种控制的效果，应该用不同的输入多次运行 predict 函数，产生的结果可能如下：

Popularity	Product quality	Affordability	Predicted number of buyers
100	0.5	0.5	2.0169999999999986
100	0.5	0.9	3.7759999999999962
100	0.9	0.5	29.21499999999997
100	0.9	0.9	53.13799999999996
10	0.5	0.5	0.7869999999999979
10	0.5	0.9	1.4769999999999976
10	0.9	0.5	3.3419999999999885
10	0.9	0.9	6.066999999999985

从上表中可以得出一些结论。购买者人数大致与产品的可承受性成正比。但是对产品质量有不成比例的依赖性：在每种流行程度和可承受性的组合中，当产品质量为 0.9 时，购买人数至少比质量为 0.5 时高出数倍。当流行程度为 100 时，倍数高达 15。流行程度与产品质量之间似乎有某种关系，当质量很高时，流行程度限制了接触到产品的人数。当质量很低时，流行程度的影响没有那么大。

在这个例子中，您已经看到 Figaro 被用作一种模拟语言。预测未来发生什么通常是模拟的工作，这也就是我们所要描述的用例。您可以简单地使用该模型进行反向推理。例如，在您将产品交给一些人并观察他们影响多少人购买产品之后，可以估计产品的质量。这是该模型有价值的备选使用方案之一。

5.5 使用马尔科夫网络

前面几节介绍的是贝叶斯网络，这种网络编码有向依赖性。现在我们将注意力转向无向依赖性。对于无向依赖性，与贝叶斯网络对应的是马尔科夫网络。我将用一个典型的图像恢复应用解释马尔科夫网络的原理。然后，我将说明如何在 Figaro 中表示这个图像恢复模型并用其进行推理。

5.5.1 马尔科夫网络定义

马尔科夫网络是概率模型的一种表现形式，包含三个部分。

- **一组变量**——每个变量有一个定义域，即该变量的可能值集合。
- **一个无向图，变量是其中的节点**——节点之间的边是无方向的，也就是说，没有从一个变量到另一个变量的箭头。这种图允许环的存在。
- **一组势**（potential）——这些势提供了模型的数值参数。我很快将详细解释这些势。

图 5-14 展示了用于图像恢复应用的马尔科夫网络。图像中的每个像素对应一个变量。图中展示了一个 4×4 的像素阵列，但是很容易推广到任何大小的图像。原则上，像素的值可以是任意颜色，但是为了例子的简单，我们用一个布尔值表示像素是鲜艳还是深色。水平或者垂直相邻的任何一对像素之间都有一条边。这些边直观地编码了这样的事实：在其他条件都相同的情况下，两个相邻的像素取相同值的可能性比取不同值的可能性更大。

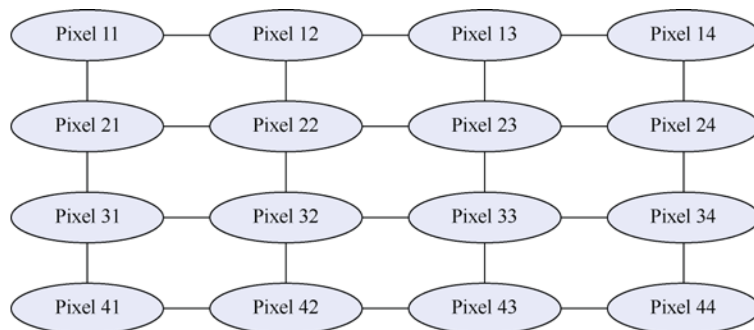


图 5-14 像素图像的马尔科夫网络

“在其他条件都相同的情况下”这个修饰语对于理解模型的含义很重要。如果忽略

两个像素之间的边，考虑每个像素的单独概率分布，那么它们实际上很可能不同。例如，根据您知道的所有其他知识，您可能相信像素 11 为鲜艳颜色的概率是 90%，像素 12 为鲜艳颜色的概率是 10%。在这种情况下，像素 11 和像素 12 不同的概率很大。但是像素 11 和像素 12 之间的边使得它们“**比其他情况下更可能相同**”，增加了它们很可能相同的知识。这一知识抵消了它们可能不同的其他知识，但是可能不会完全改变总体的结论。像素 11 和像素 12 之间的边表达的特定知识由这条边的势表示。现在，我们来看看势的定义。

势

马尔科夫网络的数值参数是如何定义的？在贝叶斯网络中，每个变量有一个 CPD。在马尔科夫网络中没有那么简单。变量没有自己的数值参数，而是在各组变量上定义所谓的**势**。存在对称依赖性时，在其他条件都相同的情况下，相互依赖的变量之间的某些组合状态比其他组合更可能出现。势指定了每个这种组合状态的权重。在其他条件相同的情况下，高权重的组合状态比低权重的组合状态更可能出现。同样，在其他条件相同的情况下，两个组合状态的相对概率等于权重的比率。

从数学上讲，势是从变量值到实数的函数，其取值只能是正实数或者 0。表 5-1 展示了图像恢复应用中单个像素上一元势的例子，表 5-2 展示了两个像素上的二元势。

表 5-1　单个像素上的一元势。这种势编码了一个事实：在其他条件都相同的情况下，某个像素“点亮”的概率为 0.4

Pixel 31	势值
F	0.6
T	0.4

表 5-2　两个相邻像素上的二元势。这种势编码了一个事实：在其他条件都相同的情况下，两个像素取相同值的可能性 9 倍于取不同值的可能性

Pixel 31	Pixel 32	势值
F	F	0.9
F	T	0.1
T	F	0.1
T	T	0.9

势函数与图的结构如何相互作用？有两条规则。

- 势函数只能提及图中相连的变量。
- 如果两个变量在图上相连，必然在某个势函数中一起提及。

在我们的图像恢复示例中，每个变量都有如表 5-1 中所示的一元势的一个拷贝，每对相邻像素（水平或者垂直）都有表 5-2 中的二元势的一个拷贝。您可以看到，势的赋

值遵循上述两条规则。

马尔科夫网络如何定义概率分布

您已经看到马尔科夫网络的定义。它如何定义概率分布？如何为每个可能世界指定概率，使所有可能世界的概率总和为 1？答案不像贝叶斯网络那么简单，但是也不是太复杂。

正如贝叶斯网络中那样，马尔科夫网络的可能世界包含所有变量的赋值，确保每个变量的值在其定义域中。这样的可能世界概率多大呢？我们用一个例子逐步加以说明。

为了简单起见，我们考虑一个 2× 2 像素阵列的如下赋值：pixel 11 = true，pixel 12 = true，pixel 21 = true， pixel 22 = false。您将观察模型中的所有势和这个可能世界的势值。一元势的值如表 5-3 所示。像素为“真”（亮）的势值为 0.4，为“假”（暗）的势值是 0.6。表 5-4 展示了来自 4 个像素组合的势值。两个像素有相同值的势值为 0.9，而其他两种情况的势值为 0.1。这两张表格涵盖了模型中的所有势。

表 5-3 可能世界一元势值示例

变 量	势 值
Pixel 11	0.4
Pixel 12	0.4
Pixel 21	0.4
Pixel 22	0.6

表 5-4 可能世界示例的二元势值

变量 1	变量 2	势 值
Pixel 11	Pixel 12	0.9
Pixel 21	Pixel 22	0.1
Pixel 11	Pixel 21	0.9
Pixel 12	Pixel 22	0.1

接下来，将所有势的值相乘。在我们的例子中将得到 0.4 × 0.4 × 0.4 × 0.6 × 0.9 × 0.1 × 0.9 × 0.1 = 0.00031104。为什么相乘？回想一下“其他条件都相同”的原则。如果两个可能世界除了一个势之外概率均相同，那么这些可能世界的概率就与它们的势值成正比。这正是将概率乘以势值时产生的效果。继续这一推理，将所有势的值相乘就得到一个可能世界的“概率”。

我在“概率”上加引号是因为这并不真的是个概率。当您这样将势值相乘时，得到的“概率”总和不为 1。这个问题很容易解决，要得到任何可能世界的概率，只需要规格化从势值乘积计算出来的“概率”。可以将这些值称为**未规格化概率**。未规格化概率的总和称为**规格化因子**，通常用字母 Z 表示。这样，将未规格化概率除以 Z 就得到概率。如果这一过程对您来说很麻烦，不要担心，Figaro 负责所有的计算。

这一讨论有个意外的发现。在贝叶斯网络中，您可以通过将相关的 CPD 条目相乘，计算出可能世界的概率。在马尔科夫网络中，您无法在不考虑所有可能世界的情况下确定任何可能世界的概率。您需要计算每个可能世界的未规格化概率，才能计算出规格化因子。因此，有些人认为表示马尔科夫网络比贝叶斯网络更难，因为解释定义概率的数值更加困难。我认为，如果您牢记“其他条件都相同”原则，就可以自信地定义马尔科夫网络的参数。您可以将规格化因子的计算留给 Figaro 完成。当然，贝叶斯网络和马尔科夫网络的参数都可以从数据中学习。根据应用程序中关系的种类，使用更适合于您的结构。

5.5.2 表示马尔科夫网络并用其进行推理

马尔科夫网络在一个方面绝对比贝叶斯网络简单：推理模式。马尔科夫网络没有诱导依赖性的概念。只要路径不被已经观察到的变量所阻塞，您就可以沿着任何路径从一个变量向另一个变量推理。如果两个变量之间有一条路径，它们就是相关的，如果一组变量阻塞了两个变量之间的所有路径，这两个变量在这一组给定的变量下就是条件独立的。

而且，因为马尔科夫网络中的所有边都没有方向，因此也没有因果或者过去与未来的概念。您通常不会考虑预测未来结果或者推理当前观测的过去原因之类的任务。作为替代，您简单地在其他变量给定的情况下推导一些变量的值。

用 Figaro 表示图像恢复模型

在图像恢复应用中，您假定观察到了一些像素，而其余像素无法观察到，希望恢复无法观察的像素。您将使用前一小节描述的模型，该模型指定了每个像素点亮的势值和相邻像素有相同值的势值。下面是表示该模型的 Figaro 代码。在 5.1.2 小节中，我曾经说过，描述对称关系有两种方法——约束方法和条件方法。下面的代码使用约束方法：

```
val pixels = Array.fill(10, 10)(Flip(0.4))

def setConstraint(i1: Int, j1: Int, i2: Int, j2: Int) {
  val pixel1 = pixels(i1)(j1)
  val pixel2 = pixels(i2)(j2)
  val pair = ^^(pixel1, pixel2)
  pair.addConstraint(bb => if (bb._1 == bb._2) 0.9; else 0.1)
}

for {
  i <- 0 until 10
  j <- 0 until 10
} {
  if (i <= 8) setConstraint(i, j, i+1, j)
  if (j <= 8) setConstraint(i, j, i, j+1)
}
```

设置每个变量上的一元约束

根据坐标设置一对变量上的二元约束

对所有相邻变量应用二元约束

下面是代码的一些说明。

- 在像素定义中，Array.fill(10, 10)(Flip(0.4))创建一个 10×10 的数组并用 Flip(0.4)的不同实例填充数组的每个元素。不同的像素都用不同的 Flip 元素定义，这很重要，因为它们都有不同的值。
- 您可能觉得奇怪，为什么对一元势使用 Flip 元素而不使用约束。对于一元势，用通常的 Figaro 方式或者约束定义效果相同。在这个例子中，Flip 得到 true 值的概率为 0.4，得到 false 值的概率为 0.6。这些概率将相乘得到可能世界的未规格化概率，和通过约束规定的一样。

  实际上，即便使用 Figaro 编码马尔科夫网络，每个 Figaro 元素也必须用某种类型的元素构造程序，以通常的方式定义。如果您的元素没有一元约束，这种常规的 Figaro 构造程序应该是中性的，不偏向任何可能世界。您可以使用 Flip(0.5)或者 Uniform 元素实现这一点。
- 在 setConstraint 的定义中，^^是 Figaro 配对构造程序。^^(pixel1, pixel2)创建一个元素，其值是元素 pixel1 和 pixel2 值的配对。
- 在 for 循环中，0 until 10 是表示整数 0～10（不含）的 Scala 标记法；换言之，表示的是整数 0,1,⋯,9。如果想要包含整数 10，应该使用 0 to 10。
- for 循环还展示了嵌套循环的一个例子。在其他语言中，这用 for 循环中的另一个 for 循环实现。在 Scala 中，可以将两个循环放在同一个 for 首标里。

用像素恢复模型推理

您希望使用像素恢复模型，根据已经观察到的像素推理无法观察的像素值。您需要 3 个条件：摄入和处理证据的手段，计算像素最可能状态的手段和查看结果的手段。我们逐个研究这些条件。

处理证据：如果您有一个 10 × 10 的像素阵列，数据可能是一个 10 × 10 的字符数组，其中每个字符是 0（表示关）、1（表示开启）或者？（表示未知）。您可以使用简单的 setEvidence 函数处理这种数据：

```
def setEvidence(data: String) = {
  for { n <- 0 until data.length } {
    val i = n / 10
    val j = n % 10
    data(n) match {
      case '0' => pixels(i)(j).observe(false)
      case '1' => pixels(i)(j).observe(true)
      case _ => ()
    }
  }
}
```

使用 Scala 模式匹配，在本例中这很像其他语言中的 switch 语句。_表示默认情况。所以对于“？”的情况没有任何观测值

计算最有可能的像素状态：这个例子介绍了前面没有考虑过的一种新查询。过去，

您希望估算元素的后验概率。这次，您感兴趣的是元素的最可能值（具有最高概率的值）。但是您对与其他元素无关的单独元素最可能值不感兴趣，而是对所有变量值的最可能组合感兴趣。您希望知道哪个可能世界的概率最高。

这种查询在 Figaro 中称为**最可能解释**（MPE）查询，因为您希望知道最可能解释数据的可能世界。MPE 查询的算法不同于目前为止您所见过的概率计算算法，但是它们是相关的。在本例中，您将使用为计算 MPE 而设计的置信传播版本。这种算法被称作 MPEBeliefPropagation。置信传播是一种迭代算法，您可以用一个参数控制迭代次数。在本例中使用 10 次迭代。您可以创建 MPEBeliefPropagation 算法的实例，告诉它以这种方式运行：

```
val algorithm = MPEBeliefPropagation(10)
algorithm.start()
```

查看结果：就是遍历所有像素，获得其最可能值并打印。您可以使用 MPEBeliefPropagation 的 mostLikelyValue 方法获得元素的最可能值。

代码如下：

```
for {
  i <- 0 until 10
} {
  for { j <- 0 until 10 } {
    val mlv = algorithm.mostLikelyValue(pixels(i)(j))
   if (mlv) print('1') else print('0')
  }
  println()
}
```

要运行这个模型，您必须提供一些输入。一般来说，这些输入从文件读入或者由另一个模块以编程方式提供。为了例子的简洁，您可以在程序中直接定义输入，如：

```
val data =
  """00?000?000
     0?010?0010
     110?010011
     11??000111
     11011000?1
     1?0?100?10
     00001?0?00
     0010??0100
     01?01001?0
     0??000110?""".filterNot(_.isWhitespace)

  setEvidence(data)
```

Scala 的"""构造程序允许创建跨越数行的字符串。您过滤所有空白字符，产生 100 个字符的字符串

在这些数据上运行时，程序输出如下：

```
0000000000
0001000010
1100010011
1100000111
1101100011
1000100010
0000100000
0010000100
0100100100
0000001100
```

马尔科夫网络就介绍到这里。本章篇幅很长，但是您已经从中学习了不少知识。现在，您知道概率模型的所有主要原理，可以为各种应用编写概率程序。接下来的几章以本章素材为基础，并对其进行详细的讲解，为您提供更多程序编写能力。下一章您将从研究 Scala 和 Figaro 容器的使用开始，构建更大、更结构化的模型。

5.6 小结

- 概率模型编码的是变量之间的关系。对称关系产生无向依赖性，而不对称关系产生有向依赖性。
- 有向依赖性从原因指向结果。因果关系多种多样。
- 贝叶斯网络使用有向非循环图编码有向依赖性。
- 贝叶斯网络中箭头的方向不一定是推理的方向。贝叶斯网络可用于网络中所有方向的推理。
- 马尔科夫网络使用无向图编码无向依赖性。
- 如果可以识别模型中变量间的关系类型并使用它们编写程序，就不会出错。

5.7 练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．对于如下每对变量，确定它们之间的依赖性是有向还是无向的，如果是有向的，说明箭头的方向。

a）玩家的扑克牌和玩家的赌注。

b）玩家 1 的扑克牌和玩家 2 的扑克牌。

c）我的情绪和今天的天气。

d）我的情绪和我是否吃过早餐。

e）起居室的温度和房屋空调的设置。

f）起居室的温度和起居室温度计的读数。

g）起居室的温度和厨房的温度。

h）新闻报道的主题和其内容。

i）新闻报道的摘要和报道的内容。

2．对如下各组变量，画出变量上的贝叶斯网络。

a）玩家 1 的扑克牌、玩家 2 的扑克牌、玩家 1 的赌注和玩家 2 跟进的赌注。

b）我起床时的情绪、上午 10 点的情绪、今天的天气和我是否吃过早餐。

c）起居室的温度、厨房的温度、房屋空调的设置、起居室温度计读数和厨房温度计读数。

d）新闻报道的主题、报道的摘要、报道的内容和对报道的评论。

3．您的任务是设计一个贝叶斯网络，建立菜汤烹制过程的模型。这个网络的目标是帮助您决定使用的材料和每种材料的数量，以及烹调中的变量（如热量和时间），以便优化不同的食品质量，如辛辣度和细腻度。

a）模型中有哪些变量？

b）绘制这些变量上的贝叶斯网络结构。

c）选择每个变量的 Figaro 函数形式。

d）在 Figaro 的函数形式中填入数值参数。

e）使用 Figaro 模型回答“在给定的材料下，您应该花费多长时间烹制菜汤，才能确保最优的细腻度”。

4．网球比赛分为一定数量的“盘”，每盘包含一定数量的“局”。先赢两盘的选手获胜。先赢 6 局的选手赢得一盘（我们忽略“平局决胜”，但是在练习 5 中您就可以建立它们的模型）。双方轮流发球，每次一局。编写一个 Figaro 程序，取得两个参数（每位选手赢得发球局的概率）并预测比赛胜者。

5．现在，细化您的网球赛模型建立单独得分的模型。在一局中，选手们努力抢分。先得 4 分的选手赢得一局，除非双方都得到 3 分，在这种情况下，先领先两分者取胜。编写一个 Figaro 程序，和之前一样取得两个参数，但是现在这些参数是每位选手在发球局中获得一分的概率。同样，您的程序预测比赛的胜者。

6．现在，进一步细化网球模型，建立每一分中连续对打的模型。选手们可以采用发球能力、球速和失误率等变量。您可以决定模型细节，包括连续对打中每次击球时选手和球的位置。

7．我的房子有由楼下的恒温调节器控制的中央空调系统。顶层通常比一楼热。创建一个马尔科夫网络，表示整个房子的温度（现在忽略恒温调节器）。编写一个 Figaro 模型表示该网络。使用该模型计算在一楼的温度为 72 华氏度时，顶楼至少为 80 华氏度的概率。

8．现在，在模型中添加恒温调节器和室外气温以及窗户是否打开。这个模型将结合有向和无向依赖性，所以不是纯粹的马尔科夫网络，但是在 Figaro 中很容易进行这样的组合。编写 Figaro 程序，用其帮助确定是否应该打开顶层的窗户。

9．考虑第 3 章的垃圾邮件过滤应用。在该应用中，每封电子邮件被单独对待。现在假定您有来自同一发件人的多封电子邮件。它们的垃圾邮件状态高度相关。

a）创建一个马尔科夫网络以捕捉这些关联性。

b）每个垃圾邮件的贝叶斯网络如图 3-8 所示。在练习 9a）中的马尔科夫网络中复制这个网络。同样，这是一个组合了有向和无向依赖性的网络。

注意：尽管我曾经说过，对象的分类决定其属性，并将这种方法用于垃圾邮件过滤器，但是在某些情况下，从反方向推理是有意义的。当所有特征总是可以观测到时（比如垃圾邮件过滤器的情况），情况就是如此。此时，明确地建立观察到的特征之上的概率分布模型可能是一种浪费。相反，可以创建一个模型，由每封电子邮件的特征决定邮件的分类。电子邮件分类与练习 9a）中的马尔科夫网络关联。这种模型称作**条件随机场**，广泛地用于自然语言理解和计算机视觉应用中。

在此，我不打算详细介绍条件随机场，但是要提醒熟悉这一模型的人们，在 Figaro 中很容易表现它。诀窍是确保观察到的电子邮件特征不是 Figaro 元素而是 Scala 变量，帮助确定表示电子邮件是不是垃圾邮件的元素之上的概率分布。当然，模型上的任何可学习参数都是 Figaro 元素，它们可以与代表决定垃圾邮件概率特征的 Scala 变量相互作用。

第 6 章　使用 Scala 和 Figaro 集合构建模型

本章介绍如下内容：

- 如何使用集合组织概率模型
- Scala 集合和 Figaro 集合之间的差别、各自的作用以及如何结合使用
- 可以用集合表达的常见建模模式，包括层次化贝叶斯建模、对象数量未知的建模情况以及在连续区间上定义的模型

在前两章中，您已经为概率建模打下了坚实的基础。本章专注于概率编程的**编程**方面，展示利用编程语言的特性帮助您构建概率模型的途径，特别关注集合。

集合是高级编程语言最有用的特性之一，因为它们能够将许多同类型的项目组织在一起，作为一个群组处理。例如，如果您要处理许多整数，可以将其放在一个数组中，然后编写一个循环读取数组中的所有条目，将其乘以 2 并加总。或者，在函数式编程中，您可以编写一个 map 函数将数组中的所有条目乘以 2，再用一个 fold 函数执行加总操作。概率编程也是如此；如果您有许多同类型的变量，可以将其放入一个集合，用 map 及 fold 等函数操作。

在 Figaro 中，上述操作可以两种方式进行。第一种是使用常规的 Scala 集合。如果您是一位有经验的程序员，无疑会熟悉数组、列表、集、映射等集合类型。Scala 提供一个丰富的集合库，帮助您组织元素和构建概率程序。

Figaro 也提供了一个集合库，添加了处理许多元素的强大功能。您可以用 Figaro 集

合完成常规 Scala 集合所不容易完成的操作。另一方面，Scala 集合提供了 Figaro 集合所不具备的功能。所以两者对概率模型都是有用的。

您将首先关注常规 Scala 集合的使用。然后研究 Figaro 集合、它们所能完成的操作以及如何、何时在一个程序中使用 Scala 和 Figaro 集合。在本章的最后两节中，您将看到 Figaro 集合的两大用处：使用可变大小数组建立对象数量未知的模型，以及在时间或者空间的连续区间上定义概率模型。

注意：因为本章专注于编程，因此包含了许多代码。本章包含 7 个完整的程序，可以在本书的代码中找到它们。熟悉 Scala 集合是很有益的，但是这些代码并没有使用特别复杂的功能。而且，您应该很熟悉第 5 章中介绍的贝叶斯网络概念。

6.1 使用 Scala 集合

目前为止看到的例子处理的是所有事物中的个例。在伦勃朗的例子中，您有一幅画，它只有一个主题、一个尺寸和一个鲜艳度。如果希望处理许多事物——例如，一个包含许多画作的模型，其中每幅画都有不止一个主题，该怎么做呢？

在编程中，您使用集合保存许多一起处理的同类事物。例如，您可能有一个数组，可以使用这个数组对其中的每个数字进行某种运算，或者加总所有数字。循环读取数组中所有数字而无需将其一一指出，可以大大缩短程序并且提供灵活性，因为您可以只在一处改变数组的大小。如果没有数组，您就必须完全展开这个循环，为同一程序中不同大小的数组使用相同的代码将变得更加困难。

使用 Scala 集合为概率编程提供了同样的好处。我将从取决于单一变量的许多同类变量开始，提供 Scala 集合多种典型应用模式的例子。

6.1.1 为依赖于单一变量的多个变量建立模型

我们来看一个经典的例子，在第 4 章中您首次遇到过这种情况。您有一个来源可疑的硬币，它可能有偏差，您不知道它正面向上和背面向上的概率，希望估算这种偏差（用它掷出正面的概率）。您还希望预测后续的掷币结果。

图 6-1 展示了这种情况的贝叶斯网络，根节点是 Bias（偏差），这是一个代表任何一次掷币出现正面的概率的实数变量。图中展示了 101 次掷币结果的节点，但是这个网络可以用于任何掷币次数。对于每次掷币，结果为正面的概率等于 Bias 变量值。您可以想象，自己将观察前 100 次掷币，并预测第 101 次掷币的结果。

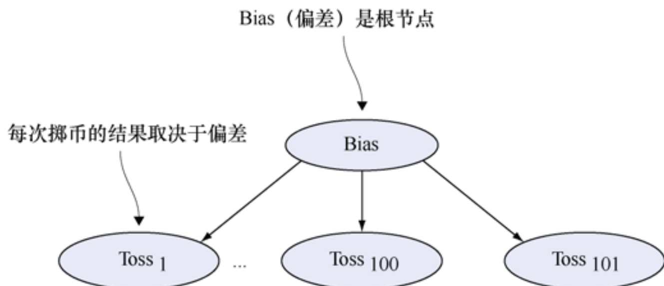


图 6-1 掷偏币的贝叶斯网络

用 Figaro 表现掷币模型

这个例子很容易通过表示所有已观察到的掷币结果的 Scala 数组，在 Figaro 中表现。我们的程序将读入表示之前观察到的掷币结果（由字符“H”和“T”组成）的命令行参数。您可以从参数中得到数组的大小，下面是模型的代码：

```
val outcomes = args(0)
val numTosses = outcomes.length

val bias = Beta(2,5)
val tosses = Array.fill(numTosses)(Flip(bias))
val nextToss = Flip(bias)
```

创建一个大小为 numTosses 的数组，其中每个项目是 Flip(bias) 的不同实例。这个数组的每个元素代表单独的一次硬币投掷

为了建立偏差的模型，您使用了一个 Beta 元素。在第 3 章中设计垃圾邮件过滤器时，使用 Beta 元素建立了单词在电子邮件中出现次数的模型。我曾经提到，Beta 元素和 Flip 及 Binomial 配合得很好，所以使用 Beta 表示硬币的偏差是合适的。Beta 和 Flip 之间关系的更多细节参见下面的补充材料。

接下来，您创建一个掷币数组，表示过去观察到的掷币结果。您使用 Scala 的 Array.fill 方法创建指定项目数量的数组，并提供每个项目的定义。对于数组中的每个项目，该定义是单独评估的。所以，每个项目是 Flip(bias)的单独实例——表示掷出正面的概率为 bias 的单独掷币结果。最后，再创建一个 Flip 元素，表示试图预测的下一次掷币结果。

共轭先验以及结合 Beta 与 Flip 的原因

在概率论中，Flip 的标准名称是伯努利分布。Beta 分布被称为伯努利分布的共轭先验。这意味着，如果您使用一个 Beta 变量和一个伯努利变量，且 Beta 变量代表伯努利变量为 true 的概率，则根据伯努利变量的结果调节 Beta 变量，可以产生另一个 Beta 变量。

现在是更仔细地观察 Beta 元素的两个参数的时候了。第一个参数α表示想象中的已见正面次数加上 1。第二个参数β与α的定义类似，但表示的是背面次数加 1。在您的例子中，使用 Beta(2,5)，表示您在学习之前想象自己已经见到 1 次正面和 4 次背面的结果。（顺便说一句，在正面和背面次数上加 1 是数学上的惯例；在下面就会看到，这使下一次掷币的预测变得更简单）这使得您可以编码这样的事实：您在看到任何数据之前，对偏差有某种先验信念。如果没有任何先验信念，可以使用 Beta(1,1)。

现在，假定观察到一次掷币的结果为正面。您可以将α递增 1 以获得后验概率。类似地，如果掷币结果为背面，则将β递增 1。一般来说，如果观测到 h 次正面和 t 次背面，则将α递增 h，将β递增 t。所以，如果先验分布为 Beta(2, 5)且观察到 3 次正面和 2 次背面，后验分布为 Beta（5,7）。这就是 Beta 分布是伯努利分布共轭先验的原因。

当偏差呈 Beta 分布时，预测下一次掷币是不是正面也很容易。发生这种情况的概率为α / (α+β)，因此对 Beta(5,7)是 5/12。

本例中的计算很简单，不需要编程语言就能完成。但是实际中的计算可能比这个复杂得多，通常无法用这么简单的公式计算后验分布并做出预测。而且，您不一定在 Figaro 中使用共轭先验。不过，在可能的情况下使用共轭先验是有意义的，因为这是表示关于某个参数先验知识的明智手段。

使用 Scala 数组使证据的陈述变得简单。outcomes 是从命令行读入的，由“H”和“T”字符表示，每个字符表示一次掷币。您可以使用如下代码观察每次掷币的对应结果：

```
for {
  toss <- 0 until numTosses
} {
  val outcome = outcomes(toss) == 'H'
  tosses(toss).observe(outcome)
}
```

用任意时间算法进行推理

您已经创建模型并陈述了证据，现在可以进行推理了。本章介绍一种进行推理的新方法，它能为您提供对推理运行时间的精确控制，那就是使用任意时间算法。顾名思义，**任意时间算法**是可以运行任意时间，产生在给定时间内最佳答案的算法。您可以告诉算法想要运行的时间，算法将尽其所能。

在 Figaro 中，运行任意时间算法和使用常规算法类似。创建该算法的一个实例并告诉它开始。Figaro 中的任意时间算法运行于单独线程，所以您可以在算法运行的同时做任何工作。一般来说，您将在算法运行期间休眠。然后，查询算法获得想要的信息。最后，杀死该算法。杀死任意时间算法很重要，因为这会释放该线程；否则，该算法将继续占用资源。

下面是在我们的掷币模型上使用重要性抽样进行推理的代码。重要性抽样既有常规

的**一次性**版本（启动之后一次性完成），也有任意时间版本。许多其他 Figaro 算法也有一次性和任意时间版本。在一次性重要性抽样中，您必须告诉算法使用的样本数量。在任意时间重要性抽样中，不需要告诉算法使用多少样本，因为它将在可用时间内取得尽可能多的样本。

```
val algorithm = Importance(nextToss, bias)
algorithm.start()
Thread.sleep(1000)
algorithm.stop()


println("Average bias = " + algorithm.mean(bias))
println("Probability of heads on next toss = " +
        algorithm.probability(nextToss, true))

algorithm.kill()
```

创建重要性抽样的一个版本，目标是查询 nextToss 和 bias。Figaro 知道这是任意时间版本，因为没有样本数量的参数

等待 1000 毫秒

释放所有资源

在参数 HTHHHHHTHTHHHTHHHHHH 下，上述代码大约运行 1 秒钟，产生如下输出：

```
Average bias = 0.6641810675997326
Probability of heads on next toss = 0.6414832927224574
```

6.1.2 创建层次化模型

本小节介绍如何使用序列，以层次化的方式扩展前面的例子。想象一下，您投掷一枚硬币，这枚硬币是钱袋中的许多硬币中的一个，这个钱袋和许多其他钱袋装在一个箱子里，以此类推。为了简单起见，您将用两级序列建立来自同一个钱袋的掷币模型，但是很容易将这一模型推广到任意级别。

在前一小节中，您已经了解掷币的属性（即正面是否向上）取决于硬币的属性（即偏差）。本小节中，硬币的偏差取决于钱袋的属性。确切地说，钱袋中的一些硬币是“公平”的，也就是出现正面的概率为 50%，而其他硬币是偏币。这个钱袋有一个属性，表示袋中硬币公平的概率。

这个例子的贝叶斯网络如图 6-2 所示。这个网络有 3 个层次，对应于层次化结构的各个级别。第一层表示钱袋，包含 FairProbability 变量。第二层表示单独的硬币及其偏差。网络中有 3 个硬币，但是很容易扩展到任意数量。第三层包含每个硬币的各次投掷。每次掷币有两个下标，表示硬币编号和掷币编号。每次掷币的结果取决于对应硬币的偏差。不同硬币投掷的次数可能不一样；在这个网络中，第 3 个硬币投掷 3 次，前两个硬币各投掷 2 次。

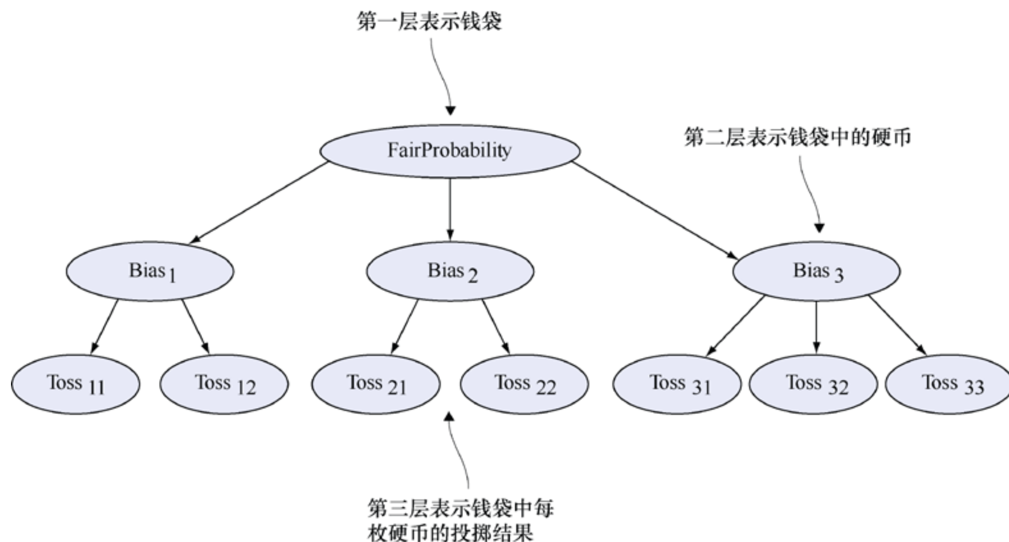


图 6-2　层次化模型的贝叶斯网络

使用两级序列可以捕捉到上述情况。第一级有两个序列，每个序列包含对应于特定硬币的 Figaro 元素：硬币是否公平，其偏差多大。对于每枚硬币，第二层中有一个序列包含代表该硬币投掷结果的 Figaro 元素。使用 for 循环很容易创建 Scala 序列。

该模型的代码如下，这段代码需要一些命令行参数，每个参数代表观察到的硬币。和前一个例子一样，每个命令行参数包含 H 和 T 字符，表示观察到的硬币投掷结果：

```
val numCoins = args.length

val fairProbability = Uniform(0.0, 1.0)

val isFair =
  for { coin <- 0 until numCoins }
  yield Flip(fairProbability)

val biases =
  for { coin <- 0 until numCoins }
  yield If(isFair(coin), Constant(0.5), Beta(2,5))

val tosses =
  for { coin <- 0 until numCoins }
  yield {
    for { toss <- 0 until args(coin).length }
    yield Flip(biases(coin))
  }
```

创建一个长度为 numCoins 的序列，其中的每个条目是一个 Flip(fairProbability)元素

对于每枚硬币，如果它是公平的，偏差为 0.5；否则，偏差为 Beta(2,5)

产生掷币序列的序列，每枚硬币一个序列

给定一枚硬币，生成 Flip(biases(coin))元素的序列，其长度为硬币投掷次数，从命令行读入

现在，可以使用一个简单的二维 for 循环陈述证据：

```
for {
  coin <- 0 until numCoins
  toss <- 0 until args(coin).length
} {
  val outcome = args(coin)(toss) == 'H'
  tosses(coin)(toss).observe(outcome)
}
```

最后，按照如下方式运行推理：

```
val algorithm = Importance(fairProbability, biases(0))
algorithm.start()
Thread.sleep(1000)
algorithm.stop()

val averageFairProbability = algorithm.mean(fairProbability)
val firstCoinAverageBias = algorithm.mean(biases(0))
println("Average fairness probability: " + averageFairProbability)
println("First coin average bias: " + firstCoinAverageBias)

algorithm.kill()
```

在参数 HTHHT H THHH HHT 下，上述代码产生如下的输出：

```
Average fairness probability: 0.7079517451620699
First coin average bias: 0.4852371044437457
```

6.1.3 建立同时依赖两个变量的模型

在前一小节中，您已经看到掷币的结果取决于硬币的偏差，而后者又取决于钱袋的公平概率。这是一个层次化组织。另一种方式是变量同时取决于两个其他变量而没有任何层次结构。对于使用 Scala 集合的前一个例子，您已经知道如何用二维数组建立这种情况的模型，其中的一个变量取决于两个其他变量，每个变量都是集合中的一个项目。例如，某个产品在某个地区的销售额可能取决于产品质量和品牌在该地区的渗透率。这里，您可以使用一个产品数组、一个地区数组和一个销售额的二维数组，每一维取决于对应的产品和地区。

注意： 本小节介绍二维数组。在现实中，您可以使用任何维数。因为这些数组是 Scala 数组，所以和常规的编程相同。

这种情况的贝叶斯网络如图 6-3 所示，理解它应该很简单。该网络有 3 个产品质量变量，2 个地区渗透率变量和 6 个销售额变量，每个对应产品质量和地区渗透率的一个组合。

尽管这个网络很简单，但是可以在其中进行很复杂的推理，特别是在观察到销售额，

试图推理产品质量和地区渗透率时。例如，产品 1 的销售额可能提供关于产品 2 质量的信息。为了了解这种工作方式，假定产品 1 在地区 1 中的销售额很高。这可能使您相信地区 1 的渗透率很高。现在，如果产品 2 在地区 1 的销售额低，您可能得出产品 2 质量低的结论。另一方面，如果产品 1 的销售额在地区 1 中的销售额也很低，您就不太倾向于得出产品 2 的质量低的结论，因为您认为销售额低的原因是地区 1 的渗透率低。

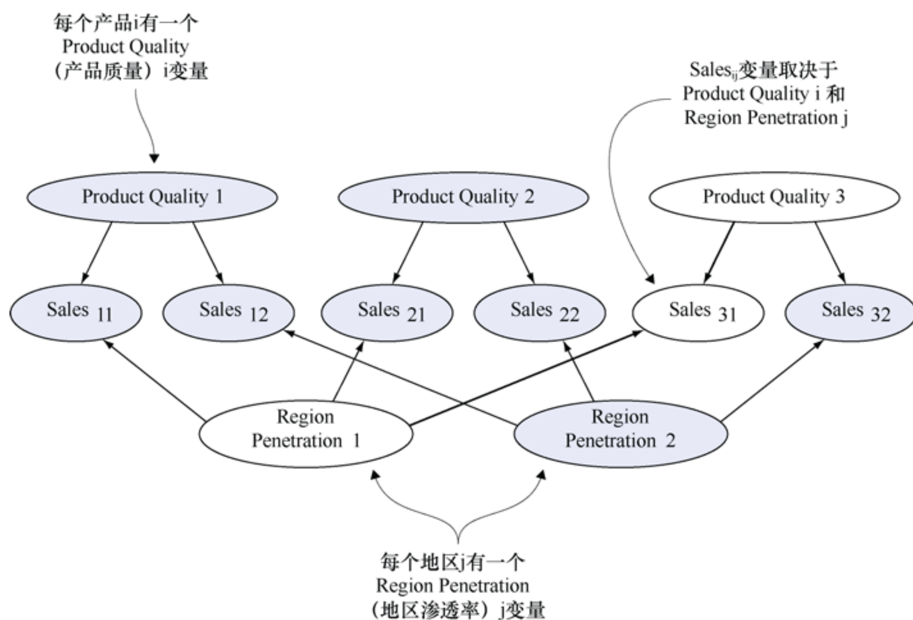


图 6-3 二维销售模型的贝叶斯网络

如果您记得第 5 章中学到的关于贝叶斯网络中推理模式的知识，就会明白路径 Sales_{11}—Region Penetration 1—Sales_{21}—Product Quality 2 是活跃的。这条路径上唯一观察到的中间变量是 Sales_{21}。但是这条路径在该变量上有会聚箭头，也就意味着路径不会在观察到该变量时被阻塞。同时，Region Penetration 1 无法观察到，该路径在 Region Penetration 1 上没有会聚箭头，所以路径也不会在那里被阻塞，综上所述，该路径是畅通的。

您可以做出结论，在这个模型中，当观察到销售额时，所有变量被连接在一起。在推理时，您将同时推理所有产品质量和所有地区的渗透率。这类推理过程称作**集体推理**，是概率编程中典型的推理类型。

Figaro 中的二维销售模型

这种情况在 Figaro 中很容易用二维数组表示。我将展示全部代码，因为您已经了解了各个成分。

程序清单 6-1　二维销售模型

```
import com.cra.figaro.library.atomic.continuous.Beta
import com.cra.figaro.language.Flip
import com.cra.figaro.algorithm.sampling.Importance

object Sales {
  def main(args: Array[String]) {
    val numProducts = args.length
    val numRegions = args(0).length

    val productQuality = Array.fill(numProducts)(Beta(2,2))
    val regionPenetration = Array.fill(numRegions)(Beta(2,2))

    def makeSales(i: Int, j: Int) =
      Flip(productQuality(i) * regionPenetration(j))
    val highSales =
      Array.tabulate(numProducts, numRegions)(makeSales _)

    for {
      i <- 0 until numProducts
      j <- 0 until numRegions
    } {
      val observation = args(i)(j) == 'T'
      highSales(i)(j).observe(observation)
    }


    val targets = productQuality ++ regionPenetration
    val algorithm = Importance(targets:_*)
    algorithm.start()
    Thread.sleep(1000)
    algorithm.stop()

    for { i <- 0 until numProducts } {
      println("Product " + i + " quality: " +
              algorithm.mean(productQuality(i)))
    }
    for { j <- 0 until numRegions } {
      println("Region " + j + " penetration: " +
              algorithm.mean(regionPenetration(j)))
    }

    algorithm.kill()
  }
}
```

命令行参数是一些字符串，每个产品一个。每个字符串有表示每个地区的一个字符，字符串的长度应该相同

Array.tabulate(numProducts, numRegions) 创建一个二维数组，其中的每个条目根据对应产品和地区索引生成。每个条目是一个 Flip，概率等于对应产品的质量乘以对应地区的渗透率

销售额观测值取自于命令行，T 表示对应产品和地区的销售额很高。对应的证据在每个 highSales 元素中陈述

创建一个任意时间重要性抽样算法，其中查询目标都是产品质量和地区渗透率元素

打印每个产品和地区的平均推理产品质量和地区渗透率

现在，您已经看到了 3 个可以用 Scala 集合表示的实用模型。是时候通过学习 Figaro

集合的使用，提高建模技术了。

6.2 使用 Figaro 集合

在前一个小节中，您学习了 Scala 集合如何帮助您创建各种涉及许多特定类型变量的有趣模型。本节介绍 Figaro 集合。Figaro **集合**是包含许多同类 Figaro 元素的数据结构。您可能认为它和 Scala 集合很类似——为什么还需要 Figaro 集合？

6.2.1 理解 Figaro 集合的用途

Figaro 集合很特殊，因为它们知道自己所包含的是元素，这些元素定义了数值之上的概率分布。它们能够让您深入元素，进行这些数值上的运算。图 6-4 展示了其工作方式。

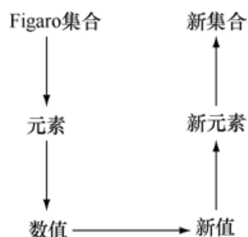


图 6-4 Figaro 集合包含元素，这些元素定义了数值之上的概率分布。Figaro 集合让您可以对这些值进行操作，用这些值产生新元素和新集合

这种能力可以帮助您完成很多有益的工作，包括：

- 如果 c 是 Figaro 整数元素集合，c.map((i: Int) => i * 2)将生成一个新的 Figaro 集合。对于原始集合中的每个元素 e，新集合中有一个等价于 Apply(e, (i: Int) => i * 2)的元素。所以，新集合由整数元素组成。每个元素对应于这样的过程：从原始集合的一个元素开始，生成其值并乘以 2。例如，如果 Figaro 集合 c 包含两个元素 Uniform(2, 3)和 Constant(5)，则 c.map((i: Int) => i * 2)是包含元素 Apply(Uniform(2, 3), (i: Int) => i * 2) 和 Apply(Constant(5), (i: Int) => i * 2)的 Figaro 集合。在实践中，这意味着该集合包含两个元素，其中一个是 4(2*2)和 6（3*2）上的均匀分布，另一个是常数 10（5*2）。
- 如果 c 是一个双精度元素的 Figaro 集合，c.chain((d: Double) => Flip(d))将产生一个新的 Figaro 集合，对于原始集合中的每个元素 e，新集合中有一个元素 Chain(e, (d: Double) => Flip(d))。第一个集合是参数的集合，第二个集合则是 Flip 集合，每个 Flip 依赖于对应的参数。例如，如果 c 包含元素 Beta(2, 1) 和 Beta(1, 2)，结果集合的两个元素将等价于 Flip(Beta(2,1)) 和 Flip(Beta(1, 2))（记得吗？复合的 Flip 是 Chain 的简写形式）。

- 如果 c 是一个 Figaro 整数元素的集合，c.exists((i: Int) => i < 0)返回一个布尔元素，表示集合中是否有包含负值的元素。类似地，c.count((i: Int) => i < 0)返回这些元素的数量。
- 您可以在容器上应用任何合并或者聚合运算。如果对 Scala 有经验，可能熟悉了 foldLeft 等运算。例如，如果 c 是一个 Figaro 整数元素的集合，可以使用 c.foldleft(_ + _)创建一个元素，代表集合中元素值的总和。举个例子，Figaro 集合 c 有两个元素 Uniform(2,3)和 Constant(5)。可能的总和有两个：2 + 5 = 7 和 3 + 5 = 8。因为第一个元素中 2 和 3 的概率各为 1/2，这两个总和的概率也各为 1/2。所以，c.foldleft(_ + _)是一个在 7 和 8 上均匀分布的元素。
- Figaro 容器上的各种操作都可以应用——详见 Scaladoc。

以上是 Figaro 集合的主要用途。其他重要的用途包括：

- 它们可以通过使用可变大小集合，表现涉及未知数量对象的情况。
- 它们还能表现涉及无穷多个变量甚至变量连续统一体的情况。

您将在本章后面学到更多有关的知识。

6.2.2　用 Figaro 集合重新实现层次化模型

作为 Figaro 集合的第一个例子，您将回到从钱袋中取得硬币进行掷币的层次化模型，看看如何用 Figaro 集合表示该模型。这个模型使用的可能是最简单的 Figaro 集合：FixedSizeArray。顾名思义，FixedSizeArray 包含固定数量的元素，每个元素以某种方式生成。如图 6-5 所示，FixedSizeArray 构造程序有两个参数：元素数量和一个元素生成器。元素生成器是一个函数，有一个表示数组中索引的整数参数，返回依赖于该索引的元素。图 6-5 中的代码创建一个包含 10 个元素的 Figaro 集合，其中第 i 个元素（从 0 开始）等于 Flip(1.0 / (i + 1))。

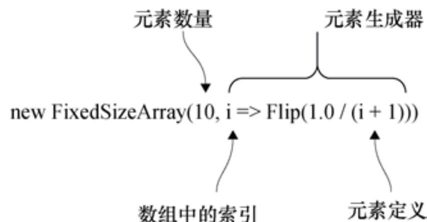


图 6-5　固定大小数组结构

因为您已经了解了这个层次化模型，我将以粗体强调变化的代码。为了说明 Figaro 容器的用法，我稍微扩展了这个例子，允许命令行参数包含？字符，表示掷币结果没有观察到。然后，您可以查询特定硬币的任何一次投掷结果是不是正面。

程序清单 6-2 使用 Figaro 容器的层次化模型

```
import com.cra.figaro.language.{Flip, Constant}
import com.cra.figaro.library.atomic.continuous.{Uniform, Beta}
import com.cra.figaro.library.compound.If
import com.cra.figaro.algorithm.sampling.Importance
import com.cra.figaro.library.process.FixedSizeArray

object HierarchicalContainers {
  def main(args: Array[String]) {
    val numCoins = args.length

    val fairProbability = Uniform(0.0, 1.0)

val isFair =
      new FixedSizeArray(numCoins + 1, i => Flip(fairProbability))

        val biases = isFair.chain(if (_) Constant(0.5) else Beta(2,5))

        val tosses =
          for { coin <- 0 until numCoins } yield
          new FixedSizeArray(args(coin).length, i => Flip(biases(coin)))

        val hasHeads =
          for { coin <- 0 until numCoins } yield
          tosses(coin).exists(b => b)

        for {
          coin <- 0 until numCoins
          toss <- 0 until args(coin).length
        } {
          args(coin)(toss) match {
            case 'H' => tosses(coin)(toss).observe(true)
            case 'T' => tosses(coin)(toss).observe(false)
            case _ => ()
          }
        }

        val algorithm = Importance(fairProbability, hasHeads(2))
        algorithm.start()
        Thread.sleep(1000)
        algorithm.stop()
        println("Probability at least one of the tosses of the third " +
                "coin was heads = " +
```

com.cra.figaro.library.process 是包含 Figaro 集合库的包

在这个固定大小的数组中，不管索引如何，每个元素都是 Flip(fairProbability)

tosses 变量是一个 Scala 序列（每枚硬币一个），其中的每个条目是一个 Figaro 集合，集合中的每个元素是一个 Flip，掷币出现正面的概率是对应硬币的偏差

对于每枚硬币，创建一个新元素，根据 isFair(coin) 的值，是 Constant(0.5) 或 Beta(2, 5)

对于每枚硬币，hasHeads(coin) 元素表示任一次掷币的结果是否正面向上。tosses(coin) .exists(b => b) 的分解参见正文

证据的观察方式几乎和以前一样。tosses(coin) 是一个 FixedSizeArray，正如在 Scala 数组中那样，您可以使用 tosses(coin)(toss) 得到一个元素

```
                algorithm.probability(hasHeads(2), true))
        algorithm.kill()
      }
    }
```

最需要解释的代码行是 tosses(coin).exists(b => b)。我们将其分解，首先，tosses 用如下代码定义：

```
for { coin <- 0 until numCoins } yield
new FixedSizeArray(args(coin).length, i => Flip(biases(coin)))
```

for…yield 结构创建一个 Scala 序列，在该序列中，每枚硬币有一个固定大小的数组。固定大小数组中元素的数量等于硬币投掷次数，这由对应的命令行参数给出。每个元素是以硬币偏差为参数的 Flip。因此，tosses(coin)是一个固定大小数组。

exists 是 Figaro 集合上的一个方法，它创建一个元素，代表集合中的元素是否有满足给定预测的值。在本例中，预测是 b => b，这是一个取得值 b 并返回相同值的函数。因为掷币的结果是布尔变量，因此 b 是布尔型值，这一预测在该元素值为 true（掷币结果为正面）时返回 true。所以，tosses(coin).exists(b => b)返回一个元素，当对应硬币任一次投掷的结果为 true 时，该元素值为 true。

现在，您已经看到了 Figaro 集合的第一个例子，下面进一步探索 Scala 集合和 Figaro 集合之间的关系。

6.2.3 结合使用 Scala 和 Figaro 集合

现在，您已经了解如何使用在元素值上定义的操作，应用 Figaro 集合。您可能觉得疑惑，Scala 集合在 Figaro 中是否还有用。它们当然有用——因为 Scala 集合的种类多于 Figaro 集合，在 Scala 集合上定义的许多方法在 Figaro 集合上并没有定义。只要您不需要使用在值上定义的操作，那么继续使用 Scala 集合可能更好。

Scala 和 Figaro 集合的相互转换

有时候，您需要同时使用两种集合。您可能需要结合 Scala 集合的灵活性和在值上进行操作的能力。幸运的是，Scala 集合很容易转换为 Figaro 集合，反之亦然。下面是 Figaro 提供的 Scala 和 Figaro 集合相互转换的一些途径。

- **Container 构造程序**——在 Figaro 中，Container 是包含有限数量元素的集合的通用类。该类提供的 Container 构造程序使用可变数量的参数，每个参数是同一值类型上的一个元素。这个构造程序返回包含这些元素的 Figaro 集合。您可以用 Container(toss1, toss2, toss3)创建包含 3 次掷币结果的 Figaro 集合。

  如果您的 Scala 集合是某种序列（它实现了 Seq 特性，如数组或者列表），可以用:_*结构将其转换为可变数量的参数。所以，如果 tosses = List(toss1, toss2, toss3)，那么 Container(tosses:_*)和前一段提到的 Figaro 集合相同。如果您的集

合是一个集（set）而不是序列，可以首先使用 toList 方法将其转换为序列，然后使用:_*结构——例如，Container(Set(toss1, toss2, toss3).toList:_*)。任何实现 Traversable 特性的 Scala 集合（包括大部分集合）可以这种方式转换为一个列表，然后转换为 Figaro 集合。

- **枚举容器中的元素**——如果您有一个 Figaro Container（有限集合），可以用 elements 方法将其转换为由元素组成的 Scala 序列（Seq）。因此，Container(toss1, toss2, toss3).elements 将返回一个包含 toss1、toss2 和 toss3 的 Seq 变量。然后，您可以将这个 Seq 转换为所需的任何 Scala 集合，如列表或者集。
- **生成从索引到元素的映射**——Figaro 集合的核心定义是从索引集中的值到元素的映射。您将在 6.4 节中更多地探索这个概念，在该节中您将考虑一个无穷过程，其中的索引集由实数组成。对于一个 FixedSizeArray，索引集由从 0 到元素数量减 1 的整数组成。对于 Container，索引集是有限的。因此，您很容易将 Container 转换为从索引到值的 Scala Map。这可以通过 Container 的 toMap 方法实现。

示例：销售额预测

为了说明如何结合使用 Scala 和 Figaro 集合，我们扩展 6.1.3 小节中的销售额示例。和以前一样，您有一些产品，每个都有各自的质量因素，这些产品在一些地区销售，每个地区的渗透率不同。您将观察前一年每种产品在各个地区的销售额，并预测来年每种产品在各个地区的销售额。您还将进一步预测公司支持每个产品线和每个地区销售力量的新雇员数。

和以前一样，您从产品和地区的一维 Scala 数组和销售额的二维 Scala 数组入手：

```
val productQuality = Array.fill(numProducts)(Beta(2,2))
val regionPenetration = Array.fill(numRegions)(Beta(2,2))
def makeSales(i: Int, j: Int) =
    Flip(productQuality(i) * regionPenetration(j))
val highSalesLastYear =
    Array.tabulate(numProducts, numRegions)(makeSales _)
val highSalesNextYear =
    Array.tabulate(numProducts, numRegions)(makeSales _)
```

这是所有产品、地区和去年及下一年销售额的完整模型，可以根据去年的销售额预测来年的销售额。为此，Scala 集合已经足够，如果不需要 Figaro 集合的额外能力，就不应该使用它们。但是，您还希望预测每种产品和每个地区新雇员的数量，因此需要 Figaro 集合。如下的代码创建一个 Scala 数组，每个产品有一个条目，每个条目是包含表示所有地区中对应产品预测销售额的元素的 Figaro Container：

```
def getSalesByProduct(i: Int) =
  for { j <- 0 until numRegions } yield highSalesNextYear(i)(j)
val salesPredictionByProduct =
  Array.tabulate(numProducts)(i => Container(getSalesByProduct(i):_*))
```

现在是用 Figaro 集合预测雇员人数的时候了。首先，对于每种产品，您将创建一个元素，表示产品在下一年中有较高销售额的地区数量。这可以用如下代码实现：

```
val numHighSales =
  for { predictions <- salesPredictionByProduct }
  yield predictions.count(b => b)
```

numHighSales 是一个 Scala 数组，每个产品有一个条目，每个条目是一个 Figaro 整数元素。salesPredictionByProduct 做成一个 Figaro 集合数组的原因是，这样对于 salesPredictionByProduct 中的每个预测集合，可以调用 predictions.count(b => b)获得代表您预测高销售量的产品数量的元素。使用其他方法很难做到这一点，而使用 Figaro 集合却很容易。

现在，对于每个产品，您将创建一个代表为该产品雇佣人数的元素，该元素依赖于产品的 numHighSales。和往常一样，Figaro 的这类依赖性用 Chain 创建。我将展示两种等价的方法，一种使用 Scala 集合，另一种使用 Figaro 集合。首先使用 Scala 集合：

```
val numHiresByProduct =
  for { i <- 0 until numProducts }
  yield Chain(numHighSales(i), (n: Int) => Poisson(n + 1))
```

上述代码创建了一个 Chain 元素组成的 Scala 数组。下面使用 Figaro 集合：

```
val numHiresByProduct =
  Container(numHighSales:_*).chain((n: Int) => Poisson(n + 1))
```

以上代码创建了一个 Chain 元素的 Figaro Container。

模型就是这样。陈述证据和运行推理的代码实际上和以前相同。但是有一个细节值得一提。因为查询目标是 numHiresByProduct 中的元素，对于 numHiresByProduct 是 Figaro Container 的版本，您需要获得其元素并将其传递给重要性抽样算法。这用如下的代码实现：

```
val targets = numHiresByProduct.elements
val algorithm = Importance(targets:_*)
```

让我们反思这个例子。Scala 和 Figaro 集合都起着重要的作用。一方面，Scala 中的多维数组很方便易用。您可以创建一个有二维索引集的 Figaro Container，但是相比起来麻烦得多。另一方面，用 Figaro 容器进行计数聚合运算很容易，但是在 Scala 中很难。实际上，您可能不得不重新实现 Figaro 的聚合逻辑，那并不是简单的工作。

现在，您已经学到了使用 Figaro 集合的基础知识，让我们来研究更为独特，但仍然实用的集合类型。

6.3　建立对象数量未知情况的模型

到目前为止，我们的 Figaro 集合都包含固定数量的对象。虽然您可以组合多个相同情况下的 Figaro 集合，但是现在它们包含的总对象数仍然是相同的。在许多情况下，对象数量是未知、可变的。对于这些情况，Figaro 提供了可变大小数组。我将首先描述可变大小数组起作用的情况，然后介绍 Figaro 的 VariableSizeArray 数据结构，最后提供一个可变大小数组的实际示例。

6.3.1　开放宇宙中对象数量未知的情况

想象您打算实现一个高速公路流量监控系统。您的目标是监控公路上的车流，预测何时出现交通拥堵。您在重要地点放置了摄像头，为了实现目标，必须完成两件事。首先，需要推算每个地点经过的车辆，包括识别何时同一辆车经过两个不同的摄像头。其次，您需要预测未来到达公路的车辆数量和造成的交通拥堵现象。

这两种任务都需要就未知数量的对象进行推理。当您推算当前在高速公路上的汽车时并不知道汽车的数量，视频图像中也没有为您提供这一数字的精确指示。有时候，图像中可能合并了很多辆车，有些车辆可能完全没有被检测到。当您预测未来的交通拥堵时，当然也不知道将会有多少车辆。

您不知道对象数量的情况称作**开放宇宙情况**（这个名称的解释参见补充材料）。开放宇宙情况的特性由两个属性描述。

- 您不知道对象的准确数量，如车辆数量，这称为**数量不确定性**。
- 您不确定对象的身份。例如，您可能不确定在不同地点的两辆车的图像是否属于同一辆车。这称为**身份不确定性**。

本小节关注数量不确定性，这种不确定性由可变大小数组解决。Figaro 也能处理身份不确定性的情况，但是您要等到下一章才能看到轻松的解决方案。

开放宇宙建模

概率编程中“开放宇宙”一词的使用来自于 Stuart Russell 和他的小组，他们在自己的概率编程语言 BLOG 中推出这一概念，这种编程语言主要设计用于表示和推理开放宇宙情况。该术语最早来源于逻辑学，在逻辑学中，“封闭世界”或者“封闭宇宙”意味着假定只有您的模型中明确提到或者从模型中直接衍生的对象才是存在的。

如果您熟悉逻辑编程语言 Prolog，就知道 Prolog 中“否定即失败”：如果不能证明某件事是真的，就认定其为假。例如，如果您试图证明图像中存在一辆绿色的车，根据已知的车辆无法证明这一点，那么就认定不存在这样的车辆。您没有必要证明没有其他未知的绿色车辆存在。否定即失败是封闭宇宙推理的一种形式。

相反，完全一阶逻辑是开放宇宙，为了证明图像中没有绿色车辆，必须证明不可能有这样的车辆，即便您对那些车辆一无所知也是如此。开放宇宙情况的建模是概率编程语言所能做到，但是在其他概率推理框架中十分困难的任务之一。

6.3.2　可变大小数组

Figaro 的开放宇宙建模方法是使用可变大小数组。可变大小数组有两个参数：一个取值为数组大小的元素，以及一个类似于固定大小数组的元素生成器。图 6-6 展示了构造可变大小数组的一个例子。VariableSizeArray 构造程序的第一个参数是一个整数元素，代表数组中的元素数量。第二个参数是元素生成器，是从数组索引到元素定义的函数。在图 6-6 中，数组中的每个元素由一个 Beta 分布定义，所以它是一个双精度元素。

构造一个可变大小数组时会发生什么？从技术上说，可变大小数组不是一个数组，而是代表着一个随机变量，这个变量的值可能是许多数组中的一个，每个数组的长度都不同，取决于数值参数的值。当您调用 VariableSizeArray 时，将创建一个代表这个随机变量的 MakeArray 元素。在可变大小数组上的所有操作都应用到这个元素。在后台，Figaro 确保尽可能高效地实现 MakeArray，以确保不同大小的数组之间达到最大限度的数据共享。特别是，较短的数组是较长数组的前缀，使用相同的元素。作为程序员，Figaro 为您完成这些工作，您无需为此担心，但是我希望确保您知道 MakeArray 是什么，因为您可能会遇到这种类型。

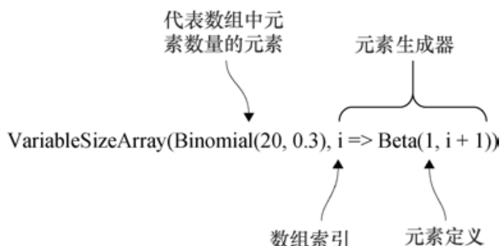


图 6-6　可变大小数组的结构

6.3.3　可变大小数组上的操作

用可变大小数组能做什么？几乎所有用固定大小数组所能做的都可以，但是两者之间存在重要的差异。在下面的例子中，vsa 表示条目为字符串元素的一个可变大小数组。下面是可变大小数组上的一些操作。

- **获得某个索引处的元素**——原则上，vsa(5)获得索引为 5 的元素（数组中的第 6

个元素）。但是这种操作很危险，您应该确定数组始终有至少 6 个元素，才能调用这个方法。否则，可能导致 IndexOutOfRangeException 异常。

- **安全地获得某个索引处的可选元素**——因为上述原因，Figaro 提供了一种更安全的方法以获得某个索引处的元素。vsa.get(5) 返回一个 Element[Option[String]]。这是一个 Scala 类型，如果索引处有一个字符串，则为 Some[String]，否则为 None。vsa.get(5)代表如下的随机过程。
  —— 根据数值参数选择元素数量。
  —— 获得合适大小的固定大小数组。
  —— 如果数组有至少 6 个元素，让 s 等于索引 5 处元素的值。返回 Some(s)。
  —— 否则，返回 None。
  可以看到，在两种情况下，随机过程都生成 Option[String]。所以该过程由一个 Element[Option[String]]表示。您可以不同方式使用一个 Element[Option[String]]，例如，可以将其传递给 Apply 元素，如果该参数是 Some(s)则进行某种有趣的操作，如果为 None 则生成默认值。
- **通过值上的函数映射可变大小数组**——这类似于固定大小数组，但是现在 Figaro 进入可变大小数组内部得到它所指向的固定大小数组，然后进入固定大小数组内部取得值。例如，vsa.map(_.length)生成一个新的可变大小数组，其中每个字符串被其长度所替代。
- **通过将值映射到元素的函数，链接可变大小数组**——这也类似于固定大小数组，Figaro 同样进入可变大小数组内部的固定大小数组，通过该函数将固定大小数组中的每个元素链接起来。例如，vsa.map((s: String) => discrete.Uniform(s:_*))创建一个可变大小数组，其中的每个元素包含来自对应字符串的随机字符。在这个例子中，s:_*将字符串转换为字符序列，discrete.Uniform (s:_*)从序列中选择随机成员。
- **折叠与聚合**——为固定大小数组提供的所有折叠和聚合也可用于可变大小数组。例如，vsa.count (_.length > 2)返回一个 Element[Int]，表示数组中长度大于 2 的字符串数量。您可以将此理解为随机地根据数值参数选择一个固定大小数组，然后计算该数组中长度大于 2 的字符串数量。同样，查阅 Scaladoc 可以了解定义的所有折叠与聚合。

6.3.4　示例：预测数量未知的新产品销售额

在这个例子中，您将想象自己正在规划公司明年的研发（R&D）投资。研发投资越多，开发出来的新产品也就越多，销售额越高。但是在做出投资的时候，您不知道指定投资水平上将开发多少新产品。

因此，您使用可变大小数组表示新产品。该模型的定义如下。

1．取得参数 rNDLevel，这是一个表示研发投资水平的双精度型变量。

2．numNewProducts 元素表示因为研发投资而开发的新产品数量，是由 Geometric(rNDLevel)定义的整数元素。Geometric 元素描述了包含一系列步骤的过程。在每个步骤之后，该过程将终止，或者进入下一步骤。进入下一步骤的概率由 Geometric 元素的参数提供（本例中是 rNDLevel）。过程的值是终止前的步骤数量。步骤数量的概率每次以 rNDLevel 为因子，以几何级数下降。rNDLevel 越高，过程继续的时间越长，开发出的新产品越多。

3．您创建一个可变大小数组 productQuality，表示每个新产品的质量。这个可变大小数组的数值参数是 numNewProducts。元素生成器是从索引 i 映射到 Beta(1,i+1)的函数。在 6.1.1 小节中，您已经学到 Beta(1, i + 1)的预期值为 1 / (i + 2)，所以产品质量倾向于随着更多新产品的开发而降低，代表了研发投资回报的逐渐下降。

4．接下来，您将产品质量转换为每个产品销售额的预测。这分两步进行。首先，使用一个以产品质量为中心的 Normal 元素生成原始销售量。但是 Normal 元素可能取负值，而销售额不可能为负数，所以在第 2 步中，您截断 Normal 元素，为其设置下界 0。这两个步骤用 VariableSizeArray 的 chain 和 map 方法完成。

5．最后，通过在 productSales 可变大小数组中折叠 sum 函数，获得总销售额。

该模型的完整代码如下：

```
val numNewProducts = Geometric(rNDLevel)
val productQuality =
  VariableSizeArray(numNewProducts, i => Beta(1, i + 1))
val productSalesRaw = productQuality.chain(Normal(_, 0.5))
val productSales = productSalesRaw.map(_.max(0))
val totalSales = productSales.foldLeft(0.0)(_ + _)
```

在这个例子中，概率编程仅用几行代码就帮助您实现了一个丰富的过程。接下来，您将学习 Figaro 集合工作的基本概念，并且观察它们是如何应用到无限集合的。

6.4 处理无限过程

本节说明如何使用 Figaro 集合定义无限空间。您的第一个问题可能是，在有限的内存中怎么可能定义元素数量无限的集合？如何在有限的时间内使用这样的集合？

答案是，这个集合中的元素只是隐含定义的。您永远不会访问无穷多个元素。但是，它们在您需要的时候可用。

警告：本节包含高级的素材。您尽可以在首次阅读时跳过它们，在本书的其余部分不需要这些素材。但是如果希望深入 Figaro 集合，体会它们的能力，就一定要返回到本节。

Process 特征（trait）是 Figaro 集合的通用表现形式，可以是有限的，也可以是无限的。下面的内容解释过程如何隐含地表现许多元素，以及如何使用过程。然后，您将看到在一系列事件点上定义的过程示例。

6.4.1　Process 特征

在 Figaro 中，**过程**（process）是在一个索引集上定义的元素集合的隐含表现形式。我所说的**隐含**，指的是给定任何索引，就能得到该索引处的元素。数组当然是这样定义的，给定上下界范围内的任何整数索引，就可以得到索引处的元素。但是对于 Figaro 过程，索引可以是您喜欢的任何类型。所以 Process 特征由两个类型参数化：索引类型和代表过程中元素取值类型的值类型。

Process[Index, Value]必须定义的方法是 generate。最简单的 generate 取得 Index 类型的索引，返回 Element[Value]。当您调用 generate 方法时，它应该创建并返回给定索引的对应元素。

使用过程时不应该直接调用 generate。前面已经看到，如果 p 是一个固定大小数组，p(5)可以得到索引为 5 的元素，没有必要调用 generate。实际上，p(5)比显式调用 generate 更安全。调用 p(5)将导致调用 generate 生成一个元素，该元素被缓存，确保每次都能得到相同的元素。如果直接调用 generate，每次调用都会得到相同索引的不同元素，这可能不是您想要的结果。

注意：在 Scala 中，p(5)是 p.apply(5)的简写。Process 特征有一个 apply 方法，该方法调用 generate 并缓存结果，使您可以调用 p(5)获得所需的元素。

generate 的另一种形式可以一次获得许多个索引处的元素。这种形式很特殊，展现了 Figaro 集合与普通集合的不同之处。在 Figaro 过程中，不同索引的元素之间可能存在依赖性。例如，假定您有一个表示某个地区不同位置降水量的过程。很明显，表示邻近位置降水量的元素相互依赖。如果一次获取一个元素，就无法捕捉这些依赖性，但是如果同时获得所有感兴趣位置的元素，就可以生成它们之间的依赖性。

第二种形式的 generate 以一个索引列表为参数。它返回 Map[Index, Element[Value]]，这是从索引到元素的一个映射。这个映射应该包含对应于参数中各个索引的元素。generate 将在后台生成表示索引对应元素之间的依赖性，但是没有放在映射中。例如，如果您有一个过程包含了不同位置的降水量，则以一组位置调用 generate 将生成表示那些位置降水量的元素，以及表示那些位置降水量之间依赖性的元素。很快您将看到这样的示例。

Process 特征还需要定义另一个方法。Index 类型的每个可能索引并不都是有相关元素的有效索引。例如，在数组中，只有从 0 到数组大小减 1 的整数是有效的索引。Process

特征有一个 rangeCheck 方法，以索引为参数，返回表示该索引是否在有效范围之内的布尔值。当您使用过程的 apply 方法获得元素时，它首先检查该索引是否在有效范围内；如果不在，则抛出 IndexOutOfRangeException 异常。

容器复习

6.2 节指出，Container 是表示 Figaro 有限集合的通用超类。现在，您已经看到了更通用的 Process 特征，应该复习一下 Container。Container 是提供特定有限索引序列的过程。只有该序列中的索引是有效的。和过程一样，Container 有可以获得一个或者多个元素的 generate 方法。

因为元素数量有限，您可以在 Container 上定义折叠和聚合。折叠操作需要遍历集合中的所有元素。这只有在集合有限的情况下才能完成。因此，通用的 Process 特征不支持折叠或者聚合。但是 Process 特征和 Container 一样支持 map 和 chain 操作。

FixedSizeArray 是 Container 的一个子类。它在两个方面很特殊。首先，有效索引的序列是从 0 到数组大小减 1 的整数。其次，FixedSizeArray 的元素被假定为相互独立。不管生成单个元素或者多个元素，每个元素都使用元素生成器生成，不生成任何编码依赖性的附加元素。

6.4.2 示例：一个健康时空过程

本章的最后一个例子使用了表现一个值在一段时间内变化的时空过程。该过程建立一位患者在一段时间内健康状况的模型。建立时空过程模型有两种方法，一是建立固定间隔的离散时间点（如每分钟），并定义一个随机变量表现每个离散时间点患者的健康状况。第二种是将时间视为连续量，在每个时间点定义一个健康变量。这使您可以访问精确时间点的健康状况变量。我将采用第二种方法，以阐述 Figaro 的无限过程。

您将定义一个 HealthProcess 对象，这是一个 Process 对象，其索引是表示时间点的双精度值，元素为表示每个时间点患者是否健康的布尔元素。HealthProcess 的声明如下：

```
object HealthProcess extends Process[Double, Boolean]
```

您需要实现 3 个方法：（1）生成单一时间点元素的 generate 版本；（2）生成多个时间点的元素及其依赖性的 generate 版本；（3）rangeCheck。

我们从 rangeCheck 开始，因为它最简单。假设这一过程有一个开始时间，定为时间 0。任何 0 或者更大的时间都是有效的。所以，rangeCheck 定义如下：

```
def rangeCheck(time: Double) = time >= 0
```

接下来，我们实现产生单一时间点患者健康状况的 generate 方法。当您孤立看待单一时间点时，健康变量是一个简单的 Flip。但是假定您不知道 Flip 的概率，希望通过学习得到。您可以引入一个参数 healthyPrior 代表这个概率。healthyPrior 和单元素 generate 定义如下：

```
val healthyPrior = Uniform((0.05 to 0.95 by 0.1):_*)
def generate(time: Double): Element[Boolean] = Flip(healthyPrior)
```

对于 healthyPrior，使用了一个从 0.05,0.15,…,0.95 中选择的离散 Uniform 元素。

现在到了最有趣的部分了。您将实现可以产生一组时间点的元素及其相互依赖性的 generate 函数。如何建立时间点间依赖性的模型？自然的模型之一是邻近的时间点可能有相同的健康值，其可能性取决于时间点之间的距离。您假定下一个时间点的影响随着时间点距离增大而以指数方式下降。您将使用参数 healthChangeRate 表示健康状况随时间变化的速度，并从数据中学习这个参数。

这样，您将创建两组元素。第一组是对应时间点的元素，用单时间点 generate 方法创建。第二组元素编码每对连续时间点之间的依赖性。您遍历排序的时间点序列，对每一对时间点创建一个元素，编码连续时间点健康状况相同的概率高于健康状况不同的概率这一事实。这一限定的强度将取决于时间点之间的距离，并由 healthChangeRate 调节：

排序时间，使我们可以始终假定它们按照顺序排列

生成从时间索引到健康状况元素的映射

```
def generate(times: List[Double]): Map[Double, Element[Boolean]] = {
  val sortedTimes = times.sorted

  val healthy = sortedTimes.map(time => (time, generate(time))).toMap

  def makePairs(remaining: List[Double]) {
    if (remaining.length >= 2) {
      val time1 :: time2 :: rest = remaining

      val probChange =
        Apply(healthChangeRate,
              (d: Double) => 1 - math.exp(- (time2 - time1) / d))
      val equalHealth = healthy(time1) === healthy(time2)
      val healthStatusChecker =
        If(equalHealth, Constant(true), Flip(probChange))
      healthStatusChecker.observe(true)

      makePairs(time2 :: rest)
    }
  }
```

按照顺序遍历各个时间点，处理每对连续的时间点

创建编码连续时间点健康状况之间依赖性的元素

```
  makePairs(sortedTimes)
  healthy
}
```

下面介绍如何创建编码连续时间点健康状况之间依赖性的元素。您将建立一个无向依赖性模型。回忆第 5 章的内容，Figaro 提供了两种编码无向依赖性的方法。使用约束的方法较为简单，但是不能从描述依赖性特征的参数中学习，而您希望学习 healthChangeRate，所以，您将使用另一种方法，创建一个布尔元素，该元素为 true 的概率取决于两个健康状态，并观察该元素是否取值 true。下面是您的输入：

- healthy(time1)——第一个时间点的健康状况。
- healthy(time2)——第二个时间点的健康状况。
- healthChangeRate——健康状况改变的速度。

上述代码创建一个名为 healthStatusChecker 的元素，您将观察它是否为 true。如果 healthy(time1) 和 healthy(time2)相等，healthStatusChecker 绝对为 true。这等于说：当 healthy(time1) 和 healthy(time2) 相等时，约束值为 1。如果 healthy(time1) 和 healthy(time2)不相等，healthStatusChecker 为 true 的概率取决于 time2 和 time1 之间的距离。直观地说，time2 和 time1 间的距离越大，健康状况越可能改变。根据 probChange 的定义，健康状况不变的概率按照 time2 – time1 除以 healthChangeRate 值的结果以指数方式下降。所以，时间差异越大，健康状况改变的概率越接近于 1。

6.4.3 使用过程

过程的定义就是如此。现在该使用它了。为此，您将假设得到了包含一些时间点健康状况的数据。您希望查询其他时间点的健康状况。您还希望查询根据数据学习到的 healthyPrior 和 healthChangeRate。

关键步骤是一次性生成感兴趣的所有时间点的元素，包括数据和查询中的时间点。那样，将创建所有必要的依赖性。如果单独为数据和查询生成元素，就无法得到它们之间的依赖性，无法使用数据预测查询时的健康状况。

这个示例使用变量消除法进行推理。变量消除法在这种模型中十分高效，是一种精确推理算法，所以是最佳的选择。但是 Figaro 的变量消除算法只在变量选择数量有限的情况下有效。这就是 healthyPrior 和 healthChangeRate 的模型使用离散、有限的选择集的原因。下面是生成必要元素、陈述证据、运行推理和得到查询答案的代码：

```
val data = Map(0.1 -> true, 0.25 -> true, 0.3 -> false,
               0.31 -> false, 0.34 -> false, 0.36 -> false,
               0.4 -> true, 0.5 -> true, 0.55 -> true)
```

时间点和观察到的健康状况映射数据

```
val queries = List(0.35, 0.37, 0.45, 0.6)
val targets = queries ::: data.keys.toList
val healthy = generate(targets)
```

为所有数据和查询点生成元素

```
for { (time, value) <- data } {
  healthy(time).observe(value)
}
```

陈述证据，generate 返回的 healthy 是从时间点到元素的映射

```
val queryElements = queries.map(healthy(_))
val queryTargets = healthyPrior :: healthChangeRate :: queryElements
val algorithm = VariableElimination(queryTargets:_*)
algorithm.start()
```

在目标元素上运行推理。您仍然使用 healthy 映射得到相关的元素

```
for { query <- queries } {
  println("Probability the patient is healthy at time " + query + " = " +
          algorithm.probability(healthy(query), true))
}
println("Expected prior probability of healthy = " +
        algorithm.mean(healthyPrior))
println("Expected health change rate = " +
        algorithm.mean(healthChangeRate))
algorithm.kill()
```

这个例子比本章前面的例子更高级。FixedSizeArray 负责 rangeCheck 和两个 generate 版本的所有定义工作。但是，如果您需要使用自定义过程或者容器，现在就已经知道怎么做了。

本章的内容就是这些，您从相对简单但很实用的设计模式开始，最终转向更强大、影响更深远的概念。在下一章中，您将学习如何使用 Figaro 创建面向对象的概率模型，进一步增强创建实用模型的能力。

6.5 小结

- 和常规编程一样，集合可以组织许多对象，使用相同代码操作所有对象。
- Scala 集合为 Figaro 元素提供实用、强大的组织结构，使您可以创建层次化或者多维依赖性模型。
- Figaro 集合提供组织元素的附加功能，同时为您提供进入元素内部操纵其值，实现折叠和量化操作的能力。
- 此外，Figaro 集合可以建立对象数量未知的开放宇宙模型。
- Figaro 过程可以在无限索引集（如时间或者空间）上建立集合模型。

6.6 练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．在您所在的城市里有 3 所小学。每所学校都有一年级到六年级的不同班级。每个班级有 30 名学生。每个学生参加数学考试。每次考试的成绩取决于学生的能力和教师的技能。编写一个概率程序以表示这种情况，从考试结果推断教师的技能。用 Scala 集合编写这个程序。

2．您打算带家人前往一个游乐园。游乐园中有 7 个设施，这些设施要么刺激，要么温和。它们有一个质量属性，可能为高、中或低。设施是刺激还是温和是可以观察到的，但是质量无法观测。您有 3 个孩子，年纪为 16、11 和 7 岁。孩子喜欢游乐设施的程度取决于年龄、设施质量以及是否刺激；大孩子更偏向于刺激的游戏，所有孩子都偏爱较高质量的设施。16 岁的孩子曾经到过这个游乐园，已经告诉您他所喜欢的游乐设施。使用 Scala 集合编写程序，预测 11 岁和 7 岁的孩子是否喜欢各种设施。

3．在高尔夫球赛中，每个洞都有一个**标准杆数（par）**，这是将球打入洞中所需的目标杆数。标准杆数通常为 3、4 或者 5。假定一位高尔夫球手的技能水平 s 决定了将球打入洞中所需的杆数。技能水平 s 是一个 0～8/13 均匀分布的实数。给定杆数的具体概率由下表定义。

par-2	par-1	par	par+1	par+2
$s/8$	$s/2$	s	$\frac{4}{5}\times\left(1-\frac{13}{8}s\right)$	$\frac{1}{5}\times\left(1-\frac{13}{8}s\right)$

创建一个数据结构表示一个高尔夫球场，球场有 18 个标准杆数各不相同的洞。编写 Figaro 程序表示选手打一场高尔夫。

a）使用该程序预测在技能水平未知的情况下，选手的成绩好于 80 杆的概率。

b）在技能水平至少为 0.3 的情况下，使用程序回答同一个查询。

c）在给定总成绩为 80 的情况下，用该程序推算选手技能水平至少为 0.3 的概率。

4．在“扫雷”游戏中，您必须找出一个网格中的地雷。网格中的每个方块可能处于如下 4 种状态之一：（1）已知有雷；（2）已知安全；（3）未知有雷；（4）未知安全。每个已知安全的方块各包含一个数字，指明水平、垂直或者对角相邻方块中地雷（已知或未知）的个数。编写一个概率程序，取得一个扫雷网格，预测每个未知方块中有一个地雷的概率。

5．您拥有一家糖果店，试图预测某一天中售出的糖果数量。假定在某一天内进店的儿童人数为 Binomial(100, 0.1)。每个孩子购买糖果的概率为 0.5,如果孩子购买糖果，数量将是 1～10 的均匀分布。使用可变大小数组预测某一天内销售的糖果总数。

6．同样，您拥有一家糖果店，但是现在孩子们一起进入商店，他们的购买决策相互影响。确切地说，如果一个孩子购买了糖果，另一个孩子就更有可能也购买糖果。每个孩子购买的糖果数量不变。假定某一天进店的孩子群体数量为 Binomial(40, 0.1)，每个群体中孩子的数量是 1～5 的均匀分布。创建一个 Figaro 过程捕捉这种情况的逻辑，并用它预测某一天的总销售数量。

第 7 章　面向对象概率建模

本章介绍如下内容：

- 使用面向对象（OO）编程技术组织复杂的概率模型
- 结合 OO 技术和关系数据库概念，使用对象和对象间关系创建灵活的模型
- 用于 OO 建模的 Figaro 结构，包括元素集合和引用
- 使用 Figaro 结构表示对象类型和关系的不确定性

在前一章，您学习了如何使用集合构造概率程序。本章继续这一主题，使用常见编程语言结构构造概率程序。本章的主题是使用面向对象编程技术表示概率模型。面向对象是一种强大的编程技术，您将会学到，它特别适合于概率建模，因为以某种情况下的对象及对象之间的关系描述这种情况很自然。Scala 是适合作为概率编程基础的编程语言，因为它既是函数式语言，又是面向对象语言。这是 Scala 被选为 Figaro 宿主语言的主要原因。

本章从基本的面向对象概念入手，因为它们适用于概率编程。您将看到类、实例、子类和继承的使用方法，以及将对象作为构件，结合这些概念编程的方法。7.2 节介绍关系概率模型，在这种模型中对象之间的关系成为核心。在开始使用关系之后，考虑关系未知的情况就很自然了。这称为**关系不确定性**，是 7.3 节的主题。您将学习实现关系不确定性表现和推理的 Figaro 功能。除了关系不确定性之外，您还将学习如何使用这些功能建立类型不确定性的模型，在这种情况下，您不能确定某个对象的类型。

本章假定您熟悉了前面几章中的概念，特别是，您应该熟悉第 2 章中的 Figaro 基础知识以及第 5 章中使用贝叶斯网络和马尔科夫网络建立依赖性模型的知识。此外，您应该熟悉 Java 或者 Scala 等语言中的面向对象编程功能。一般来说，我不使用 Scala 特有的功能（如特征），但是将使用 Scala 对象，所以如果您熟悉 Java，应该能够理解本章中的面向对象编程结构。

7.1　使用面向对象概率模型

您可能很熟悉 OO 编程和它所带来的好处，但是我希望具体介绍我认为最重要的好处。在本章的过程中，我将强调这些好处如何应用到概率编程。

为了了解使用 OO 技术进行概率编程的动机，我们回到第 1 章的角球示例。提醒一下，我们将建立足球角球的模型，考虑攻方和守方的技能、环境条件（风力等）等信息。这种情况使用 OO 方法很自然。球员是对象，属于某个球队，后者也是对象。球员完成移动或者踢球等动作。不同类的球员（如中锋和守门员）可以用 Player 类的子类建模。球员和其他球员以及球互动，球也是对象。环境也可以作为对象，和球以及球员互动。

面向对象编程有两个主要好处：

- **为复杂程序提供结构**——对象是捕捉一组数据和行为的内聚单位。对象为这些数据和行为提供了统一接口，其内部结构是封装的，不为程序其余部分所见。这使得程序员可以模块化的方式修改对象的内部结构，而不会影响程序的其余部分。例如，完整的守门员模型可能包含许多变量和细致的依赖性，但是其中只有少数几个（如扑救技术）可能影响到模型的其余部分。
- **实现代码重用**——首先，相同的类代码以及内部结构可以供所有类实例重用。相同的球员模型可用于同一个球队的许多球员。其次，继承性实现了不同类共同特征的重用。例如，中锋和守门员共享许多变量和依赖性。

这些优势自然适用于概率编程。但是面向对象甚至更加适合于概率模型。在概率编程中，我们构建真实世界的模型，真实世界可以自然地用对象描述。

本节首先讨论类、实例、属性和子类等基本概念，以及它们在概率建模中的应用。然后介绍两个基于第 5 章中打印机示例的例子。第一个例子说明如何将扁平的贝叶斯网络转换成面向对象模型。第二个例子说明如何重用代码，建立同一网络中不同种类打印机的模型。

7.1.1　理解面向对象建模的元素

在面向对象程序中，您有描述一般数据和行为的类和那些类的实例，这些实例包含了具体数据和行为的实例化。例如，您可能有一个通用的打印机**类**，以及作为该类**实例**的特定打印机。打印机类描述了打印机的一般行为，而打印机实例有特定于该打印机的

数据和行为，这些行为从类的行为衍生而来。

注意：**对象**一词有时候指的是类，有时指的是实例。Scala 也有 object 关键字，用于定义单态类（只有唯一特殊实例的类）的实例。因此，我从现在起避免使用**对象**而使用含义更清晰的**类**和**实例**。

对于概率程序来说，这意味着：

- **概率类模型定义生成随机变量值的通用过程**。例如，打印机类模型描述生成电源是否开启、是否卡纸、打印机总体状态等的通用过程。
- **实例是通用类模型的特定实例化，描述生成属于这个具体实例的随机变量值的一个过程**。例如，打印机实例使用其类模型描述生成这台特定打印机电源是否开启、是否卡纸和打印机总体状态值的过程。

要在 Figaro 中表示概率类和实例，可以使用常规的 Scala 类及实例。您可以给出值为元素的 Scala 类属性。例如，在表示打印机的 Scala 类中，可以有属性 powerButtonOn、paperFlow 和 state。每个属性都定义为一个元素。在 Scala 中，类属性代表通用定义。类的每个实例都有由这个元素定义的特殊属性。例如，如果您有一个名为 myPrinter 的打印机实例，它就有属性 myPrinter.powerButtonOn、myPrinter.paperFlow 和 myPrinter.state。这些属性是定义为 Figaro 元素的常规 Scala 变量。所以，您在 myPrinter 的属性值上定义了一个生成过程。下面的代码片段展示了包含这些属性的 Printer 类定义，以及名为 myPrinter 的特定实例定义。myPrinter.powerButtonOn 是一个 Flip(0.95)元素，myPrinter.paperFlow 是一个 Select 元素，等等。

```
class Printer {
  val powerButtonOn = Flip(0.95)
  val paperFlow = Select(0.6 -> 'smooth, 0.2 -> 'uneven, 0.2 -> 'jammed)
  val state = // etc.
}
val myPrinter = new Printer
```

另一种表达方式是，一组概率类定义提供了随机过程的通用定义，这些随机过程可以在不同实例上重用多次。这和常规的 OO 编程类似，类模型可以用于许多不同的实例。如果您有打印机、网络和软件类，可以重用不同的打印机、网络和软件实例。这些实例可以采用您所需要的任何配置。例如，可以有用不同软件工作的不同打印机。甚至在同一个程序中使用同一个类的不同实例，例如一个网络上的多台打印机。

因为 Scala 支持子类和继承，您也可以对 Figaro 模型使用子类和继承。这使您得以重用多个类定义的不同部分。例如，您可以有不同种类的打印机，如激光打印机和喷墨打印机。不同类的打印机可以共享相同的行为，如卡纸的倾向，同时也可以有不同的行为，如激光打印机特有的墨粉耗尽现象。熟悉的子类、继承和重载概念使您能够轻松地表现此类情况。

7.1.2　重温打印机模型

上面已经介绍了一些基本概念，现在我们来看看它们在实际中的应用。您将使用第5章中的打印机程序。图7-1重现了这一问题的贝叶斯网络。

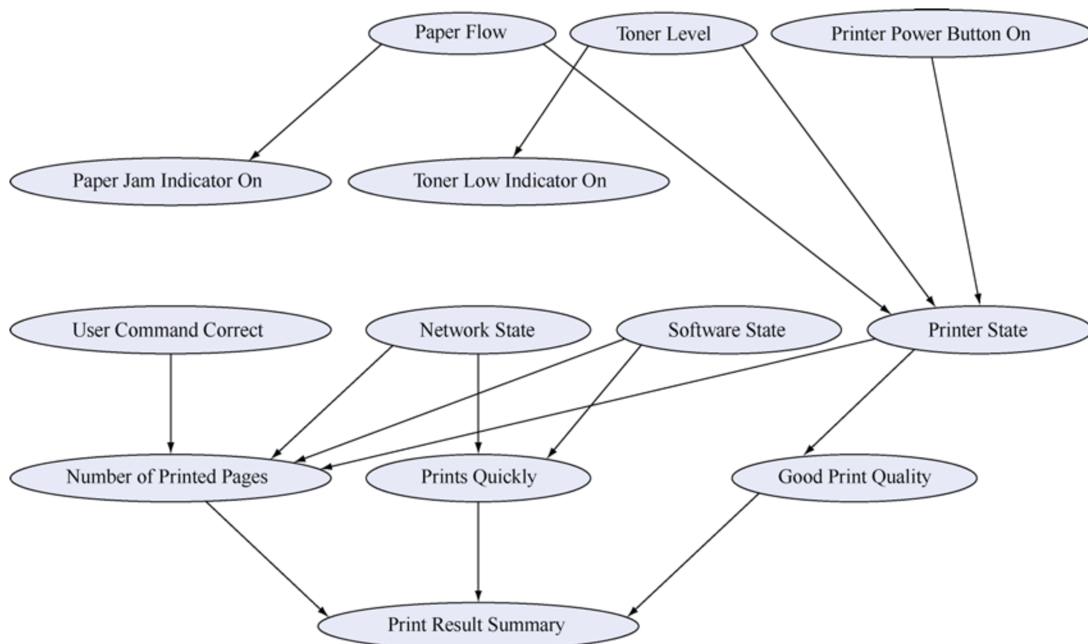


图7-1　从第5章重现的打印机贝叶斯网络。注意，这个网络是“扁平”的，所有变量都处于同一级别，没有任何结构。直观地说，网络的上半部分对应于打印机的属性，但是这个结构在网络中并不明显

贝叶斯网络描述了因为打印机问题而致电服务台的用户。该用户描述了打印结果的摘要，这是已打印页面数量、打印速度是不是很快以及打印质量的函数，而这些因素又取决于用户命令和计算机网络、软件及打印机的状态。然后，贝叶斯网络进入打印机的更多细节，建立了电源按钮是否开启、进纸状态和墨粉水平的模型。

这个模型是面向对象技术的自然候选。首先，您有打印机。网络中的6个变量与打印机相关，唯一与模型中其余部分相关的变量是Printer State（打印机状态）。如果使用一个打印机类，其余变量都被封装在类中。同样，虽然在这个特殊网络中用户、网络和软件只有一个节点，但是这些节点可能更复杂，包含某种封装的内部状态。最后，您可以有一个表现总体打印体验的对象。

您可以集中所有相关的变量，将打印机模型转换为面向对象的贝叶斯网络，如图7-2所示。在这个网络中，附属于打印机的6个变量被集合为Printer类。同样，与打印体验

相关的 4 个节点被放入 Print Experience 类。我已经选择将用户、网络和软件放入各自的类中。尽管在这个特殊模型中每个类都只有一个变量，但是将这些节点放入单独的类，使您能够在稍后修改类，添加更多结构，而不会破坏总体模型。这是使用面向对象风格的常见理由。

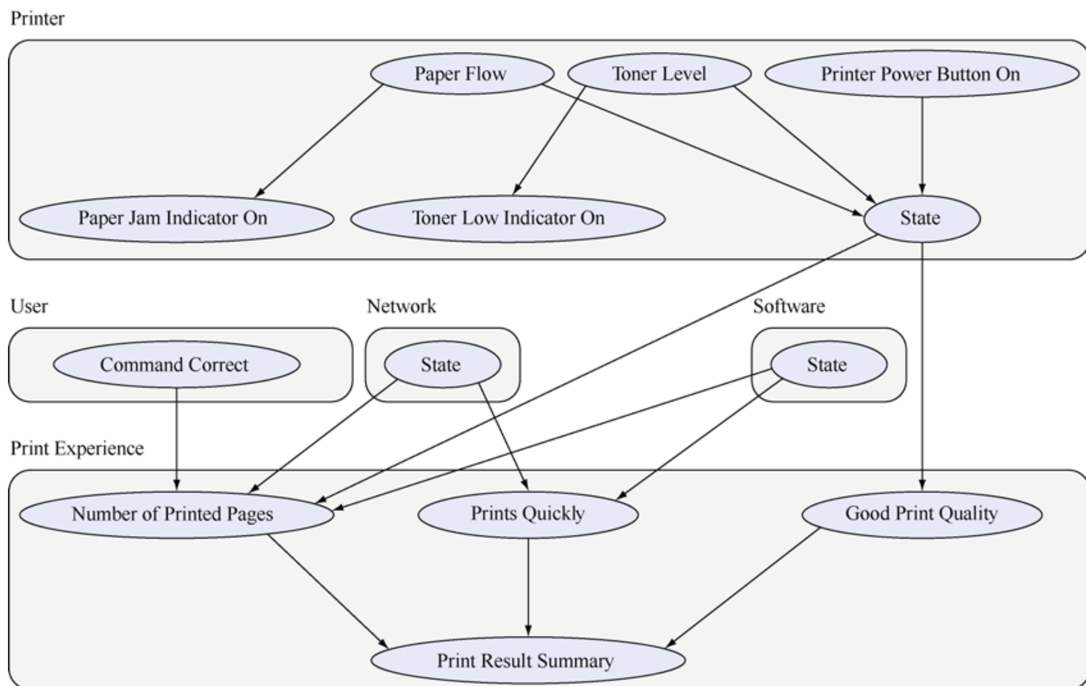


图 7-2 打印机贝叶斯网络的面向对象表现形式。带有阴影的方框组合了关联的变量。方框顶部的标签是类名称。注意这个模型是如何提供图 7-1 的贝叶斯网络中不明显的结构的

以面向对象方式表现打印机问题的代码将在下面介绍。代码很简单，这种风格对使用过面向对象编程的任何人来说都应该很熟悉。对概率编程来说，没有太多新内容。这段代码展示了一个三步的过程。

1．定义类模型。

2．创建类实例。

3．用这些实例进行推理。

在这个例子中，实例已经用于在打印体验摘要给定的情况下，推断打印机电源按钮是否开启，但是它们可以用于支持任何推理模式，就像常规的贝叶斯网络一样。

这个问题的代码清单分为 3 个部分。第一部分表示打印机、软件、网络和用户的类模型。除了变量现在放入类定义之外，这些类模型和以前一样。下面是代码。

程序清单 7-1　用面向对象方法解决打印机问题：类模型

```
package chap07

import com.cra.figaro.language._
import com.cra.figaro.library.compound._
import com.cra.figaro.algorithm.factored.VariableElimination

object PrinterProblemOO {
  class Printer {
    val powerButtonOn = Flip(0.95)
    val tonerLevel = Select(0.7 -> 'high, 0.2 -> 'low, 0.1 -> 'out)
    val tonerLowIndicatorOn =
      If(powerButtonOn,
         CPD(tonerLevel,
             'high -> Flip(0.2),
             'low -> Flip(0.6),
             'out -> Flip(0.99)),
         Constant(false))
    val paperFlow = Select(0.6 -> 'smooth, 0.2 -> 'uneven, 0.2 -> 'jammed)
    val paperJamIndicatorOn =
      If(powerButtonOn,
         CPD(paperFlow,
             'smooth -> Flip(0.1),
             'uneven -> Flip(0.3),
             'jammed -> Flip(0.99)),
         Constant(false))
    val state =
      Apply(powerButtonOn, tonerLevel, paperFlow,
            (power: Boolean, toner: Symbol, paper: Symbol) => {
              if (power) {
                if (toner == 'high && paper == 'smooth) 'good
                else if (toner == 'out || paper == 'jammed) 'out
                else 'poor
              } else 'out
            })
  }

  class Software {
    val state = Select(0.8 -> 'correct, 0.15 -> 'glitchy, 0.05 -> 'crashed)
  }

  class Network {
    val state = Select(0.7 -> 'up, 0.2 -> 'intermittent, 0.1 -> 'down)
  }

  class User {
    val commandCorrect = Flip(0.65)
  }
```

接下来是 PrintExperience 类。该类使用前面的 4 个类作为其参数，在其定义中引用了那些类内部的变量。

程序清单 7-2　打印机问题：PrintExperience 类

打印机体验涉及特定打印机、软件、网络和用户，所以 PrintExperience 类以这些类的实例为参数

打印体验取决于打印机、软件、网络和用户的特定属性。这些属性通过加上实例的名称和“.”引用，例如，“printer.state”引用 printer 实例的 state 属性

```
class PrintExperience(printer: Printer, software: Software, network:
   Network, user: User) {

  val numPrintedPages =
    RichCPD(user.commandCorrect, network.state, software.state,
   printer.state,
        (*, *, *, OneOf('out)) -> Constant('zero),
        (*, *, OneOf('crashed), *) -> Constant('zero),
        (*, OneOf('down), *, *) -> Constant('zero),
        (OneOf(false), *, *, *) -> Select(0.3 -> 'zero, 0.6 -> 'some, 0.1
   -> 'all),
        (OneOf(true), *, *, *) -> Select(0.01 -> 'zero, 0.01 -> 'some, 0.98
   -> 'all))
  val printsQuickly =
    Chain(network.state, software.state,
          (network: Symbol, software: Symbol) =>
            if (network == 'down || software == 'crashed) Constant(false)
            else if (network == 'intermittent || software == 'glitchy)
   Flip(0.5)
            else Flip(0.9))
  val goodPrintQuality =
    CPD(printer.state,
        'good -> Flip(0.95),
        'poor -> Flip(0.3),
        'out -> Constant(false))
  val summary =
    Apply(numPrintedPages, printsQuickly, goodPrintQuality,
          (pages: Symbol, quickly: Boolean, quality: Boolean) =>
          if (pages == 'zero) 'none
          else if (pages == 'some || !quickly || !quality) 'poor
          else 'excellent)
}
```

在定义了所有类之后，您可以创建这些类的实例并关联它们。这定义了一个生成实例属性值的过程；换言之，一个特定的概率模型。然后，您可以提供证据，提出关于具体属性的查询。

程序清单 7-3　打印机问题：创建实例并向模型提出查询

创建类的实例。myExperience 通过 PrintExperience 构造程序与 myPrinter、mySoftware、myNetwork、me 关联

```
val myPrinter = new Printer
val mySoftware = new Software
val myNetwork = new Network
val me = new User
val myExperience = new PrintExperience(myPrinter, mySoftware,
   myNetwork, me)
```

```
def step1() {
  val answerWithNoEvidence =
   VariableElimination.probability(myPrinter.powerButtonOn, true)
  println("Prior probability the printer power button is on = " +
   answerWithNoEvidence)
}

def step2() {
  myExperience.summary.observe('poor)
  val answerIfPrintResultPoor =
   VariableElimination.probability(myPrinter.powerButtonOn, true)
  println("Probability the printer power button is on given a poor result =
   " + answerIfPrintResultPoor)
}

def main(args: Array[String]) {
  step1()
  step2()
}
```

查询和证据涉及实例的特定属性

基本打印机模型就是这样。接下来，我们加入相同情况的多台打印机，细化该模型。

7.1.3 关于多台打印机的推理

目前，您已经看到面向对象是如何用于构造模型、提供封装的好处的。面向对象编程的主要优势是实现了代码重用。它有两种重用机制：同一模型中相同类的多个实例，以及同一类的多个定义稍有不同的子类。

本小节介绍如何使用 Scala 实现这两种重用机制。首先，我们来看看同一个类的多个子类的定义。想象一下，您有两类打印机：激光打印机和喷墨打印机。两种打印机都有电源按钮和进纸装置，但是每种打印机各有自己的特性。您可以使用标准的子类和继承获得想要的效果。

程序清单 7-4 提供 Printer 类及其两个子类（LaserPrinter 和 InkjetPrinter）的代码。Printer 是一个抽象类，其中的 state 属性没有定义，由子类提供。这个类定义了 3 个所有打印机共有的属性。这种设计有两个好处。首先，它实现了 LaserPrinter 和 InkjetPrinter 之间的代码重用。其次，它定义了打印机的共用接口，模型的其余部分可以依赖这些接口。如果 myPrinter 是 Printer 的某个子类的实例，它可以依赖类定义得到 state 属性。

程序清单 7-4 Printer 类层次结构

```
abstract class Printer {

  val powerButtonOn = Flip(0.95)
  val paperFlow =
    Select(0.6 -> 'smooth, 0.2 -> 'uneven, 0.2 -> 'jammed)
  val paperJamIndicatorOn =
    If(powerButtonOn,
       CPD(paperFlow,
           'smooth -> Flip(0.1),
           'uneven -> Flip(0.3),
           'jammed -> Flip(0.99)),
       Constant(false))

  val state: Element[Symbol]
}

class LaserPrinter extends Printer {
  val tonerLevel = Select(0.7 -> 'high, 0.2 -> 'low, 0.1 -> 'out)
  val tonerLowIndicatorOn =
    If(powerButtonOn,
       CPD(tonerLevel,
           'high -> Flip(0.2),
           'low -> Flip(0.6),
           'out -> Flip(0.99)),
       Constant(false))

  val state =
    Apply(powerButtonOn, tonerLevel, paperFlow,
          (power: Boolean, toner: Symbol, paper: Symbol) => {
            if (power) { //
              if (toner == 'high && paper == 'smooth) 'good
              else if (toner == 'out || paper == 'jammed) 'out
              else 'poor
            } else 'out
          }
    )
  }

  class InkjetPrinter extends Printer {
    val inkCartridgeEmpty = Flip(0.1)
    val inkCartridgeEmptyIndicator =
      If(inkCartridgeEmpty, Flip(0.99), Flip(0.3))
    val cloggedNozzle = Flip(0.001)

    val state =
      Apply(powerButtonOn, inkCartridgeEmpty,
            cloggedNozzle, paperFlow,
            (power: Boolean, ink: Boolean,
             nozzle: Boolean, paper: Symbol) => {
               if (power && !ink && !nozzle) {
                  if (paper == 'smooth) 'good
                  else if (paper == 'uneven) 'poor
                  else 'out
```

这是个抽象类,因为它包含没有定义的状态属性

所有打印机共用的代码

声明所有打印机都有的属性的公共接口。具体的子类必须实现这个属性

特定于 LaserPrinter 子类的属性

特定于 Inkjet Printer 子类的属性

公共接口的单独实现

```
            } else 'out
          }
    )
}
```

公共接口的单独实现

以上是创建共享某些代码和公共接口的多对象模型的方法。因为 LaserPrinter 和 InkjetPrinter 都是同一个 Printer 类的子类，LaserPrinter 和 InkjetPrinter 的实例可以作为 PrintExperience 的参数，后者的预期参数为 Printer。实际上，模型中可以有两个 PrintExperience 实例，一个用于 LaserPrinter，另一个用于 InkjetPrinter。这两个实例可以共享相同的用户、网络或者软件。下面是它们的定义：

```
val myLaserPrinter = new LaserPrinter
val myInkjetPrinter = new InkjetPrinter
val mySoftware = new Software
val myNetwork = new Network
val me = new User
val myExperience1 =
  new PrintExperience(myLaserPrinter, mySoftware, myNetwork, me)
val myExperience2 =
  new PrintExperience(myInkjetPrinter, mySoftware, myNetwork, me)
```

现在，您可以进行和这两种体验的公共元素有关的推理，这些元素包括软件、网络和用户。下面是一个例子：

```
def step1() {
  myExperience1.summary.observe('none)
  val alg =
    VariableElimination(myLaserPrinter.powerButtonOn, myNetwork.state)
  alg.start()
  println("After observing that printing with the laser printer " +
    "produces no result:")
  println("Probability laser printer power button is on = " +
    alg.probability(myLaserPrinter.powerButtonOn, true))
  println("Probability network is down = " +
    alg.probability(myNetwork.state, 'down))
  alg.kill()
}
def step2() {
  myExperience2.summary.observe('none)
  val alg =
    VariableElimination(myLaserPrinter.powerButtonOn, myNetwork.state)
  alg.start()
  println("\nAfter observing that printing with the inkjet printer " +
    "also produces no result:")
  println("Probability laser printer power button is on = " +
    alg.probability(myLaserPrinter.powerButtonOn, true))
  println("Probability network is down = " +
    alg.probability(myNetwork.state, 'down))
  alg.kill()
}
```

运行上述程序，将打印如下结果：

```
After observing that printing with the laser printer produces no result:
Probability laser printer power button is on = 0.8573402523786461
Probability network is down = 0.2853194952427076

After observing that printing with the inkjet printer also produces no result:
Probability laser printer power button is on = 0.8978039844824163
Probability network is down = 0.42501359502427405
```

上述结果背后的逻辑是，观察到喷墨打印机也无法打印之后，情况和激光打印机出问题的时候就不太可能一样了（如果是那样，则意味着喷墨打印机也出现故障），更有可能的是其他方面（如网络）的问题。所以，激光打印机的电源按钮开启的概率上升，网络中断的概率也增大。

最后，我们想象您不知道所拥有的打印机类型。您知道它是激光打印机或者喷墨打印机，但不确定是哪一种。这称作**类型不确定性**，需要额外的处理机制，您将在 7.3 小节中学习。但是，在进入类型不确定性的处理之前，您必须将面向对象模型推广到具有一般关系的模型。这些模型称作**关系概率模型**，是下一小节的主题。

7.2 用关系扩展 OO 概率模型

想象您试图建立社会化媒体（如 Facebook）网站上用户的模型，希望通过观察帖子和评论推断他们的兴趣和关系。根据社会化媒体的定义，它们是关系式的，本质就是人们之间的关系。在人们及其帖子与评论之间也存在自然的关系。此外，帖子和帖子的评论之间也有某种关系。这一领域有自然的 OO 结构，包括人、帖子和评论的类，但是为了成功地建立该领域的模型，必须使用关系概率模型。

关系概率模型就是一种 OO 模型，其中的关系明确，成为表现形式的核心部分。在过去几年中，已经开发了多种关系概率模型语言，如概率关系模型和马尔科夫逻辑。本节不介绍任何特定的语言，而是介绍一种组合了许多语言特性的通用编程风格。

为了描述如何设计和实现关系概率模型，我将采取三个步骤，首先，我将介绍如何用类概率模型描述通用模型。然后，介绍如何用这个通用模型描述具体情况。最后，介绍如何用 Figaro 实现这些步骤。

7.2.1 描述通用类级模型

在关系概率模型中，类概率模型有两个目的。

- 描述模型结构，包括模型中的类，类属性和类之间的关系。
- 定义支配概率模型的概率依赖性、函数形式和数值参数。

通过观察模型结构，很容易理解关系概率模型。图 7-3 展示了我们的社会化媒体应

用的模型结构。

- 该模型有 4 个类：Person（人）、Connection（联系）、Post（帖子）和 Comment（评论）。
- 类包含属性。属性有两类。

  —— 简单属性用椭圆表示，表示某种类型值上的随机变量。例如，Person 有一个 Interest（兴趣）属性，这是一个字符串上的随机变量。Post 有一个 Topic（主题）属性，也是字符串上的随机变量。Connection 有 Type（类型）属性，可以是家人、朋友或者熟人。Comment 有一个 Match（匹配）属性，这是一个布尔变量，表示评论者的兴趣和帖子主题之间是否存在匹配。

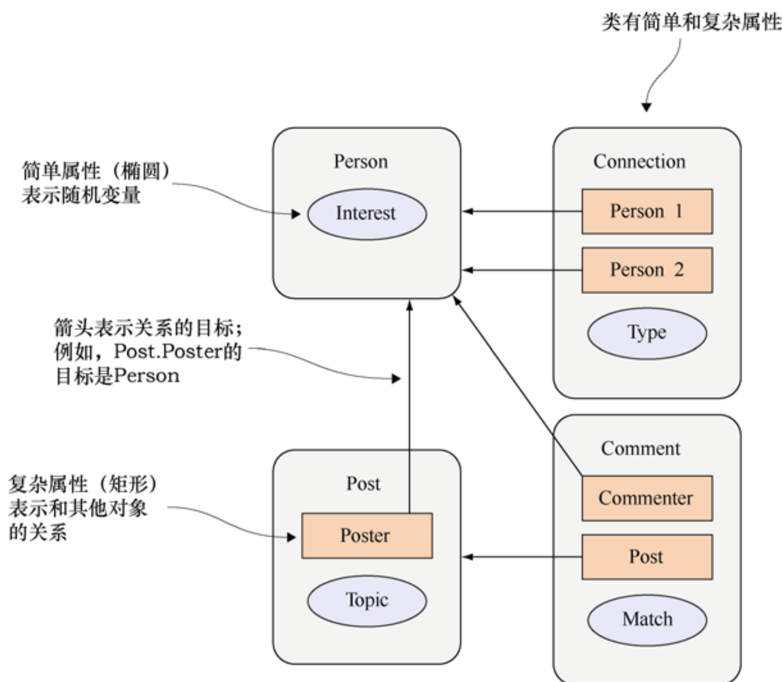


图 7-3　关系概率模型结构

  —— 复杂属性用矩形表示，定义和其他对象的关系。例如，Post 类的 Poster 属性自然地表示发表帖子的人。对于 Post 的任何实例，Poster 属性的值将是 Person 实例。因此，Poster 属性定义了 Post 实例和 Person 实例之间的关系。

有了关系结构之后，下一步是定义表现形式的概率部分。您需要做的主要工作是定义依赖性。我们首先考虑实例级别的依赖性。指导方针是，**实例的一个属性可能依赖该实例的其他属性，或者相关实例的属性**。例如，如果您有 Post 的特定实例 Post 1 和 Person 实例 Amy，Post 1.Poster 的值是 Amy，Post 1.Topic 可能依赖于 Amy.Interest。

上述的依赖性是实例级的。那么，如何在类级别上编码依赖性呢？很简单！只需要从 Person 类的 Interest 属性画一个指向 Post 类的 Topic 属性的箭头。图 7-4 展示了社会化媒体模型的概率依赖性。概率依赖性与图 7-3 的关系结构有很多重叠，所以可以清晰地理解它们。这个例子包含了有向依赖性（以粗箭头表示）和无向依赖性（以粗虚线表示）。

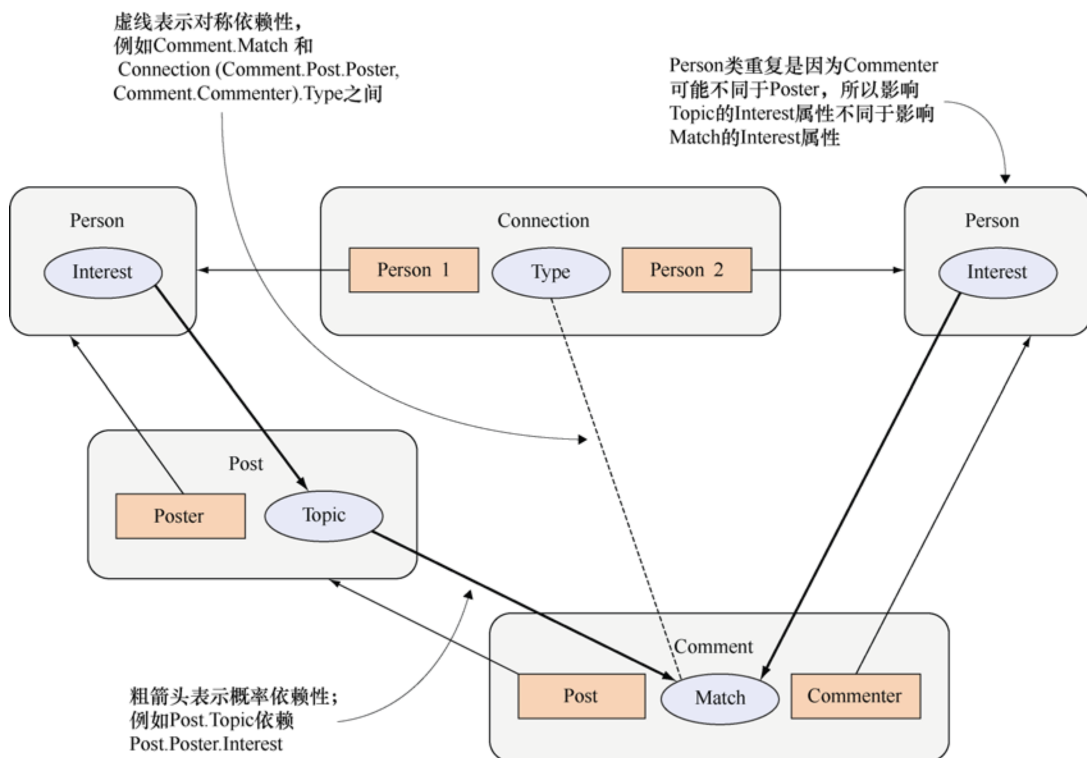


图 7-4 社会化媒体关系概率模型中的概率依赖性

规定了依赖性之后，可以规定函数形式和数值参数。这方面没有什么特别之处，您可以使用之前使用过的方法。回忆第 5 章可以得知，贝叶斯网络中的有向依赖性用条件概率分布描述，而马尔科夫网络中的无向依赖性则以势编码。势是在一组变量上定义的函数，为变量值的每个组合指定一个非负实数值。

我们的例子情况如下。

- 因为 Person.Interest 没有父变量，您将得到可能兴趣上的一个概率分布。
- Connection.Type 同样没有父变量，您将得到可能的联系类型之上的一个概率分布。
- 对于 Post.Topic，您有一个给定 Post.Poster.Interest 的 CPD，指定帖子的主题通

常与发帖者的兴趣相匹配。

- 对于 Comment.Match，您有给定 Comment.Post.Topic 和 Comment.Commenter.Interest 的 CPD。确定性的 CPD 说明，如果帖子主题和评论者的兴趣一致，就存在一个匹配。
- 在评论是否匹配以及发帖者和评论者之间存在的联系类型上，您有一个势。联系类型由 Connection 的 Type 类型标识，其中 Person 1 是 Comment.Post.Poster，Person 2 是 Comment.Commenter。您将这个属性称作 Connection(Comment.Post.Poster, Comment.Commenter).Type。对于 Comment 的任意实例 *c*，可以确定两个 Person 实例：*c*.Post.Poster 和 *c*.Commenter。然后，确定联系两个人的 Connection 实例，获得 Connection 的 Type 属性。所以，势在该属性和 *c*.Match 上定义。这个势说明如果联系类型是熟人，Comment 很有可能匹配，但是如果联系类型是亲密的朋友，概率就有所下降，是家人时更低。

关系概率模型的类级定义就到此为止。下一步是创建实例，指定它们之间的关系。我将说明这一步中定义实例属性上概率分布的方式。

7.2.2　描述某种情况

为了描述某种情况，您需要准确指定该情况下每个类的实例及其联系方式。例如，在我们的社会化媒体模型中，情况可以由表 7-1 说明。

表 7-1　社会化媒体中的实例规格及其相互之间的关系。第一个表格说明了 Person 的 3 个实例。第二个表格说明了 Post 的 3 个实例以及每个实例的 Poster 复杂属性值。第三个表格说明了 Comment 的 4 个实例以及每个实例的 Post 和 Commenter 属性值

Person
Amy
Brian
Cheryl

Post	Poster
Post 1	Amy
Post 2	Brian
Post 3	Amy

Comment	Post	Commenter
Comment 1	Post 1	Brian
Comment 2	Post 1	Cheryl
Comment 3	Post 2	Amy
Comment 4	Post 3	Cheryl

您不需要说明 Connection 的实例，因为它们自动衍生。对于 Comment 的每个实例 *c*，自动衍生 Connection(*c*.Post.Poster,*c*.Commenter)。对于 Comment 1，Post.Poster 是 Amy，Commenter 是 Brian，所以衍生出 Connection(Amy, Brian)。类似地，对于 Comment 2 和 3，可以衍生出 Connection(Amy, Cheryl) 和 Connection(Brian, Amy)。

对于 Comment 4，您也能衍生出 Connection(Amy, Cheryl)。这个 Connection 实例和 Comment 2 衍生出的相同。这是一个重点。您希望从 Cheryl 对 Amy 的两个帖子的评论推断出某些情况。如果为每个评论使用不同的 Connection 实例，就无法从评论的组合推

断联系类型。使用相同的 Connection 实例，就能够确保 Cheryl 对 Amy 帖子的所有评论都可用于推理联系的类型。（另一方面，在我们的模型中 Connection(Amy, Brian) 与 Connection(Brian, Amy)不同。这是一种设计上的选择，您已经选择将联系视为不对称关系。将联系当成对称关系的模型很容易建立，在这种情况下，Comment 1 和 Comment 3 都使用相同的 Connection 实例 ）

得出某种情况的概率模型

使用类概率模型的主要好处之一是某种情况的概率模型可以自动衍生，您不需要自己动手。但是重要的是理解其工作原理。以通常的方式进行，首先，您确定描述情况的变量。接下来，指定依赖性。最后，指定描述这些依赖性特征的函数形式和数值参数。

对于变量，在实例和关系指定之后，它们自动为情况中的所有实例定义一组简单属性，每个属性都成为模型中的一个变量。在这个例子中，有如下属性：

- Amy.Interest, Brian.Interest, Cheryl.Interest
- Post 1.Topic, Post 2.Topic, Post 3.Topic
- Comment 1.Match, Comment 2.Match, Comment 3.Match, Comment 4.Match
- Connection(Amy, Brian).Type, Connection(Amy, Cheryl).Type, Connection(Brian, Amy).Type

对于每个属性，您可以从通用类级模型确定它们所依赖的属性。这就可以构建一张表现该情况下依赖性的图，如图 7-5 所示。例如，根据 Person 类的模型，Person.Interest 没有父变量，所以 Amy.Interest、 Brian.Interest 和 Cheryl.Interest 也是如此。根据 Post 类模型，Post.Topic 的父变量是 Post.Poster.Interest。这意味着，在实例级别上，Post 1.Topic 的父变量是 Amy.Interest，Post 2.Topic 的父变量是 Brian.Interest，Post 3.Topic 的父变量也是 Amy.Interest。

现在，我们来考虑 Comment 1.Match。根据类模型，Comment.Match 有两个父变量 Comment.Post.Topic 和 Comment.Commenter.Interest。此外，它有一条到 Connection (Comment.Post.Poster, Comment.Commenter).Type 的无向边。对于特定实例 Comment 1，Comment 1.Post 是 Post 1，Comment 1.Commenter 是 Brian，Comment 1.Post.Poster 是 Amy。因此，Comment 1.Match 有两个父变量：Post 1.Topic 和 Brian.Interest。它还有一条到 Connection(Amy, Brian).Type 的无向边。在贝叶斯网络中可以看到这些边，这和 Comment 的其他实例类似。

严格来说，依赖性的图不是贝叶斯网络也不是马尔科夫网络，因为它包含有向和无向边。但是，理解图中定义的概率模型应该相当容易。图中包含有向边的部分可以理解为常规的贝叶斯网络。无向边对网络加以额外的约束，正如 Figaro 的约束一样。实际上，下一小节中也就是这么在 Figaro 中实现的。

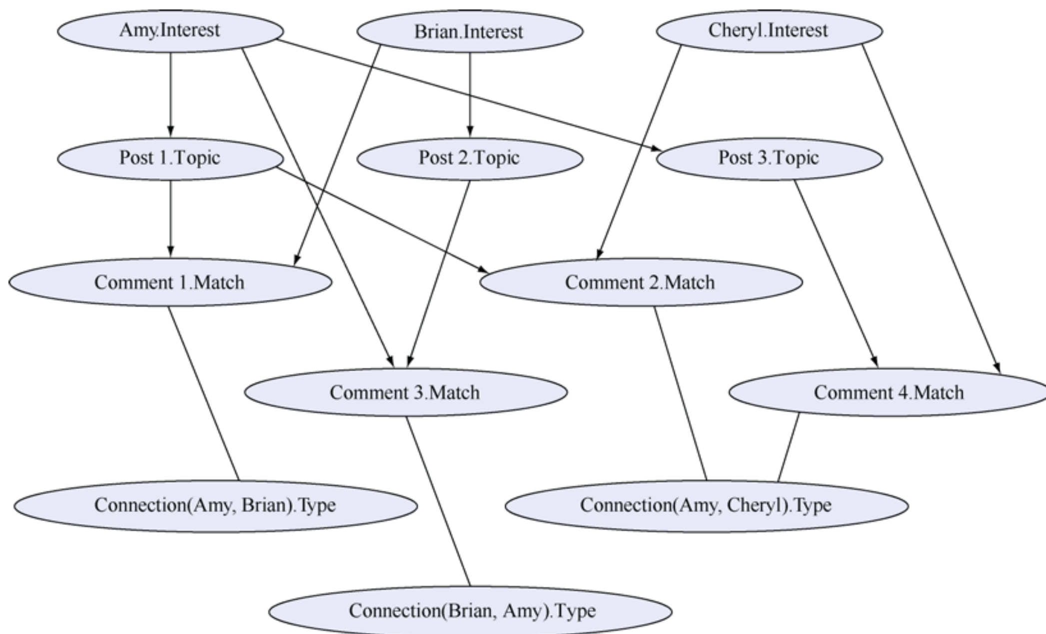


图 7-5 社会化媒体示例的依赖性图。这个网络用表 7-1 的情况说明，从图 7-3 的类级模型中自动构造。对于每个类级的简单属性，网络中有一组变量，类的每个实例使用一个。可以想象，当您的实例较多时，网络很快就会蔓延

理解了依赖性图之后，该情况的数值参数可以很自然地从类级模型中得出。例如，给定 Amy.Interest 情况下的 Post 1.Topic CPD 可以从给定 Post.Poster.Interest 的 Post.Topic 类级 CPD 中得出。类似地，Comment 1.Match 和 Connection(Amy, Brian).Type 上的约束可以从 Comment.Match 和 Connection(Comment.Post.Poster, Comment.Commenter).Type 的类级势上得出。

此时，您已经有了定义该情况下所有实例简单属性值上的组合概率分布所需的所有成分。简言之，关系概率模型是描述某种情况下实例属性上的概率模型的一种手段。它与直接构建图 7-4 所描述的模型相比有两个主要的好处。这也是本章开始时强调的面向对象技术的两个主要好处。

- 通过用对象及关系描述应用领域，为潜在的复杂情况提供了结构。图 7-4 中的图不错，但是它只涉及 3 个用户、3 篇帖子和 4 条评论。想象一下，如果有数以千计的用户、帖子和评论会是什么情况。试图构建和维护这样的网络将是一项恐怖的工作。但是我们在图 7-3 中提供的类级模型不管有多少实例都保持不变。确实，您必须像表 7-1 中那样说明实例和关系，但是这个任务要容易得多了。许多关系数据库工具可以帮助您组织许多实例和它们之间的关系。

- 类模型可以应用到具有不同关系的许多实例。这是强大的重用机制，因为您可以将概率模型的相同类级定义应用到许多情况。

好了，现在您已经知道了关系概率模型的定义以及如何定义概率分布，是时候了解如何使用概率编程实现一个模型了。

7.2.3 用 Figaro 表现社会化媒体模型

用 Figaro 表现一个关系概率模型很容易。您已经从 7.1.2 小节中的面向对象打印机模型了解了这方面的概念，社会化媒体模型也很类似。

首先，和以前一样创建具有属性的类模型。在我们的例子中，您可以为类设置复杂的属性参数。代码如下：

```
class Person() {
  val interest = Uniform("sports", "politics")
}
class Post(val poster: Person) {
  val topic = If(Flip(0.9), poster.interest, Uniform("sports", "politics"))
}
```

高级用法注释: 在我们的例子中，您可以使 poster 成为 Post 的一个参数，因为帖子依赖于发帖者，但是发帖者不依赖于帖子。所以您可以生成发帖者，然后生成帖子。如果发帖者也依赖于帖子，就不能将 poster 作为 Post 的参数，因为发帖者无法在帖子之前生成。对此有两种解决方案。一种是使用 Scala 的惰性求值，这样帖子可以依赖于发帖者，但是发帖者的属性在需要之前不会求值。另一种方法是为 Post 添加一个内部属性 poster，该属性最初为 null，在调用 Figaro 推理算法之前设置这一属性值。只要对 poster 的所有引用发生在 Chain 元素内部（在我们的例子中是 If 内部，这也是 Chain 在语法上提供的一种便利），在推理之前就不需要 poster 元素，到那个时候，该属性的值将被正确设置。

您可以类似的方式定义其他两个类。需要注意的是 Connection 类。联系从两个人衍生而来。您希望能够调用 connection(person1, person2)获得两个人之间的联系，必须确保在多次调用 connection(person1, person2)时得到的都是相同的联系。Figaro 提供了一种简单的方法：使用 com.cra.figaro.util 包中的 memo 函数，该函数确保一个函数在用相同参数调用时都返回相同值。下面是该函数的使用方法：

```
import com.cra.figaro.util.memo
class Connection(person1: Person, person2: Person) {
  val connectionType = Uniform("acquaintance", "close friend", "family")
}
def generateConnection(pair: (Person, Person)) =
  new Connection(pair._1, pair._2)
val connection = memo(generateConnection _)
```

对 Comment，使用有向和无向依赖性的组合。之前您已经了解了使用这种组合的方

法。下面是代码。===（3 个等号）是 Figaro 的相等运算符，这是一个布尔元素，如果两个参数相等，元素值为 true。

```
class Comment(val post: Post, val commenter: Person) {
  val topicMatch = post.topic === commenter.interest
  val pair =
    ^^(topicMatch, connection(post.poster, commenter).connectionType)
  def constraint(pair: (Boolean, String)) = {
    val (topicMatch, connectionType) = pair
    if (topicMatch) 1.0
    else if (connectionType == "family") 0.8
    else if (connectionType == "close friend") 0.5
    else 0.1
  }
  pair.addConstraint(constraint _)
}
```

接下来，提供一些证据：

```
post1.topic.observe("politics")
post2.topic.observe("sports")
post3.topic.observe("politics")
```

最后，回答一些查询：

```
println("Probability Amy's interest is politics = " +
  VariableElimination.probability(amy.interest, "politics"))
println("Probability Brian's interest is politics = " +
  VariableElimination.probability(brian.interest, "politics"))
println("Probability Cheryl's interest is politics = " +
  VariableElimination.probability(cheryl.interest, "politics"))
println("Probability Brian is Amy's family = " +
  VariableElimination.probability(connection(amy, brian).connectionType,
                                  "family"))
println("Probability Cheryl is Amy's family = " +
  VariableElimination.probability(connection(amy, cheryl).connectionType,
                                  "family"))
println("Probability Cheryl is Brian's family = " +
  VariableElimination.probability(connection(brian, cheryl).connectionType,
                                  "family"))
```

这个程序打印如下输出：

```
Probability Amy's interest is politics = 0.9940991345397325
Probability Brian's interest is politics = 0.10135135135135132
Probability Cheryl's interest is politics = 0.7692307692307692
Probability Brian is Amy's family = 0.5472972972972974
Probability Cheryl is Amy's family = 0.4205128205128205
Probability Cheryl is Brian's family = 0.3333333333333333
```

注意，Amy 最有可能对政治最感兴趣，因为她发表了两篇关于政治的帖子。Cheryl

也可能对政治感兴趣，尽管她没有发帖，但是对 Amy 关于政治的帖子发表了评论。另一方面，Brian 发表了关于体育的帖子，所以他可能对政治不感兴趣，不过，他对 Amy 的一篇政治帖子发表了评论，由于他对政治不感兴趣，所以有可能是 Amy 的家人。最后，因为 Cheryl 从未对 Brian 的帖子发表评论，Cheryl 是 Brian 的家人的概率和最初一样——1/3。

7.3　建立关系和类型不确定性的模型

我们来详细介绍一下我们的社会化媒体示例。假定在任何给定时间，某些主题很热门，人们发表这方面的帖子的可能性最高。您可以用主题对象代替字符串来建立这种模型，创建一个有 hot（热度）属性的 Topic（主题）类。Post 类的 topic 属性现在变成了一个指向 Topic 类的复杂属性。这很不错。

现在，假定您不知道帖子 p 的主题。您可能打算使用自然语言处理系统识别主题，但是算法不完善。您不确定 p.topic 的值，这是一个复杂属性。换言之，您不确定帖子和主题之间的关系，后者可能是 Topic 的实例之一。这种情况称为**关系不确定性**，有时候也称作**引用不确定性**，因为您不知道 p.topic 对象引用哪一个对象。

类似地，回到打印机示例，您可能不知道所拥有的打印机类型。您可能在服务台工作，打进电话的用户没有告诉您所使用的打印机类型。从技术上说，这意味着 PrintExperience 对象的打印机参数未知。这种情况称作**类型不确定性**。类型不确定是关系不确定性的特例，因为您可以创建每个可能类的假想打印机（不确定哪一个是用户所指的打印机）来处理这种情况。

关系不确定性和类型不确定性的概念很容易理解，但是在 Figaro 中需要一些额外的机制才能处理。这种机制称作**元素集合和引用**，有多种用途。在本章中，您将看到最常见的用途——处理关系不确定性。在下一章中，您将看到另一种重要用途——处理动态模型。本节首先介绍元素集合和引用的基础知识，然后介绍如何实现具有未知主题的社会化媒体示例，最后介绍具有类型不确定性的打印机示例。

7.3.1　元素集合和引用

您可能还不知道，每个 Figaro 元素都有一个名称，并属于某个元素集合。**元素集合**是一个 Figaro 集合，其中的元素由一个字符串（名称）索引。您不知道这一点的原因是，默认情况下，您不需要指定元素的名称和元素集合。默认情况下，元素的名称是一个空串（“ ”）。而且，除非另做说明，每个元素都属于一个默认的元素集合。目前，您还没有理由为元素取一个特殊的名称，或者将其放在默认集合之外的任何元素集合，所以我没有介绍这些概念。但是对于关系不确定性，我们需要这些概念。

元素集合很有用，它们为您提供了不通过 Scala 变量确认元素的方法。如果 ec 是一个元素集合，您可以用 ec.get(n)获得与名称 n 相关联的元素。有时候，这是获得元素句柄的唯一方法。当您编写自己的程序时，在需要时可能不知道名称为 n 的是哪一个元素。这正如编译时和运行时类型检查一样。有些时候，您在编译时无法完全确定某个变量的类型，所以必须进行运行时检查。同样，在编译时您可能不知道如何获得一个特定元素，元素集合为您提供了在运行时动态获得变量的手段。

加以思考，就可以得到处理关系不确定性的一种方法。您不确定 Post 的 topic 属性值。topic 所指的主题有一个名为 hot 的属性，这是一个元素。您想要做的就是从 Post 类内部引用 topic.hot。但是用 Scala 做不到这一点，因为 topic 不是 Topic 的特定实例！作为替代，您可以使 Post 和 Topic 成为元素集合，并使用 get("topic.hot")获得需要的元素。

现在，您可能会问，topic.hot 不是一个名称，如何做到这一点？这是两个名称的连接。第一个名称 topic 是 Post 元素集合中的元素名称。在本例中，这是一个 Element[Topic]，其值表示帖子的特定主题。第二个名称 hot 是 Topic 元素集合中一个元素的名称。这种名称的连接称作**引用**。图 7-6 展示了这一引用的解析方法。重点是引用中每个后续的名称在一个元素集合中定义，该集合是前一个元素的一个可能值。

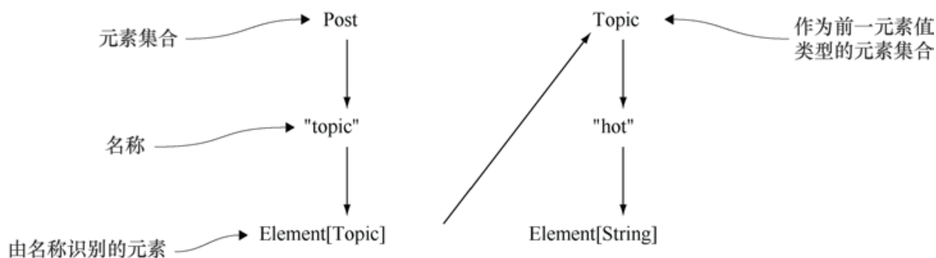


图 7-6　解析 topic.hot 引用

如何理解引用？让我们考虑一下定义随机过程的原则方法。假定有两个可能的主题 sports 和 politics，它们都是 Topic 类的实例。Topic 是一个元素集合，它拥有一个名为“hot”的属性 hot。（一般来说，Figaro 的属性名称不一定要与 Scala 变量的名称相同，但是两者相同往往是有意义的。）post1 是 Post（也是一个元素集合）的一个实例，具有 poster 和 topic 属性。主题属性的名称为“topic”。调用 post1.get("topic.hot")定义一个随机过程，该过程的步骤如图 7-7 所示。

还记得 Figaro 的元素表现的是随机过程吗？您可以使用一个元素表示 post1.get("topic.hot")定义的随机过程。在元素集合中取得一个引用正是这样实现的，因为它返回表示随机过程的元素。您可以在模型中使用这个元素，就像使用其他任何元素一样。Figaro 在后台负责找出引用和所有可能的解，而您完全不需要为此担心，这真是一件好事。

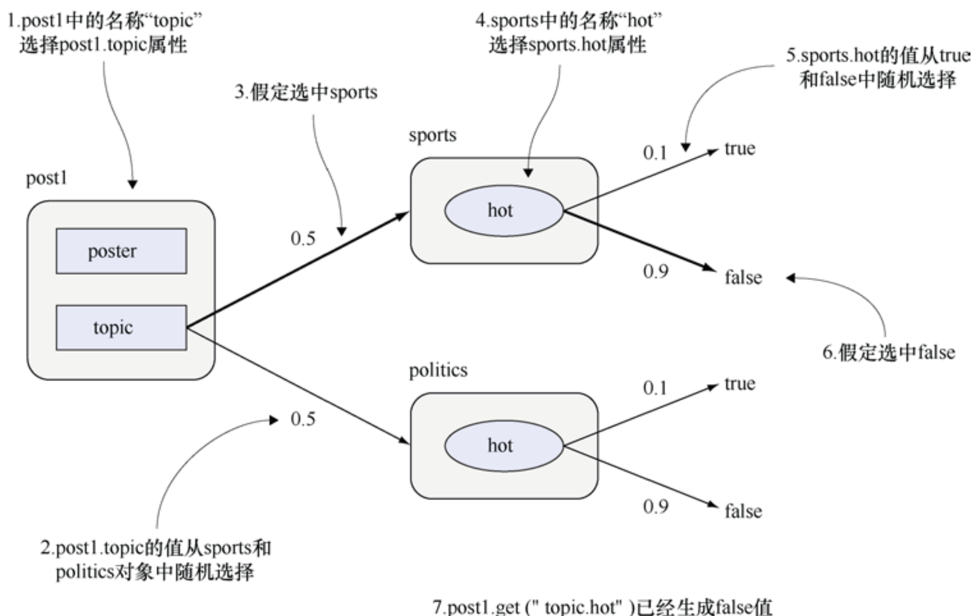


图 7-7 引用是生成值的随机过程。在这个框图中，过程为元素集合中的引用 topic.hot 生成值

有一个微妙的要点值得一提。如果 Topic 类也有一个名为 important 的属性，这个属性和名为 hot 的属性之间存在依赖性，该怎么办？现在假定您创建一个表示 post1.get("topic.hot")的元素和另一个表示 post1.get("topic.important")的元素。在任何给定的可能世界中，topic 是 sports 或者 politics；它无法同时为这两个值。在任何可能世界中，不管得到的是 important 还是 hot 元素，topic 在 post1 中的解也必须是相同的结果。这就引入了 post1.get("topic.hot")和 post1.get("topic.important")之间的依赖性。这是您的直觉中可以预计到的，Figaro 的算法正确地处理这一情况。

7.3.2 具有关系不确定性的社会化媒体模型

有了上述的机制，您就为再次处理社会化媒体模型做好了准备，这次模型在帖子的主题上具有不确定性。首先创建一个 Topic 类：

```
class Topic() extends ElementCollection {
     val hot = Flip(0.1)("hot", this)
}
```

第一行说明，Topic 的实例也是一个元素集合，意味着元素可以指定名称并放在集合中。第二行创建一个 Flip(0.1)元素，给定名称 hot，并将其放在元素集合 this 中。在

Topic 类的上下文中，this 关键字指的是 Topic 的特定实例。所以，Topic 的任何给定实例都是一个元素集合，该元素集合中有一个名为 hot 的元素，这是一个 Flip(0.1)。

这是您第一次遇到 Flip 构造程序的两个额外参数。这些参数是可选的，通常可以忽略。只在需要为元素取名，或者将其放在特定元素集合中时才需要它们。但是如果您只想为元素取名，或者只想将其放在特定的元素集合中，还必须指定其他可选参数。Flip 不是提供这些可选参数的唯一构造程序。一般来说，所有 Figaro 内建元素构造程序都可以指定一个名称和元素集合。

现在，您可以定义 Topic 类的两个实例：

```
val sports = new Topic()
val politics = new Topic()
```

Person 有一个 interest 属性，这是一个未知的 Topic 实例。interest 属性是一个 Element[Topic]：

```
class Person() {
  val interest = Uniform(sports, politics)
}
```

Post 也是一个元素集合，它有一个名为 topic 的属性，该属性是依赖于 poster.interest 的 Element[Topic]：

```
class Post(val poster: Person) extends ElementCollection {
  val topic =
    If(Flip(0.9), poster.interest,
       Uniform(sports, politics))("topic", this)
}
```

现在，Comment 类中使用了引用 topic.hot。具体的逻辑是，即使人们对热门的主题不感兴趣，也将为之撰写评论。所以，如果主题与评论者的兴趣相符或者是热门话题，那么就可能有一篇评论。下面是 Comment 类的相关代码：

```
class Comment(val post: Post, val commenter: Person) {
  val isHot = post.get[Boolean]("topic.hot")

  val appropriateComment =
    Apply(post.topic, commenter.interest, isHot,
    (t1: Topic, t2: Topic, b: Boolean) => (t1 == t2) || b)
  // add the undirected constraint on appropriateComment and the connection
  // type as before
}
```

如果您观察了之前的 isHot 定义就会注意到，必须指定元素的值类型（这里是布尔类型）。实际上，您通常必须指定值类型。仅从元素名称，Scala 编译器无从知晓元素的

值类型。但是为了能够使用 isHot，您必须知道其值类型。在方括号中指定值类型作为 get 的类型参数，告诉 Scala 编译器所需的信息。

上面的代码已经涵盖了整个模型。您可以和以前一样创建实例和关系——这里不再重复代码。我将展示如何提供关于复杂属性的证据。这和简单属性完全一样，您可以编写 post1.topic.observe(politics)观察 topic 属性的值是 Topic 实例 politics。

您还可以查询复杂属性的值。这个模型支持一些有趣的查询和推理。我们从下面的查询开始，这些查询说明，您可以查询人们的兴趣和帖子的主题，这些都是 Topic 类的实例：

```
println("Probability Amy's interest is politics = " +
  VariableElimination.probability(amy.interest, politics))
println("Probability post 2's topic is sports = " +
  VariableElimination.probability(post2.topic, sports))
println("Probability post 3's topic is sports = " +
  VariableElimination.probability(post3.topic, sports))
println("Probability Amy is Brian's family = " +
  VariableElimination.probability(connection(brian, amy).connectionType,
                                  "family"))
```

上述代码打印如下结果：

```
Probability Amy's interest is politics = 0.9656697011762803
Probability post 2's topic is sports = 0.24190044937009825
Probability post 3's topic is sports = 0.06958146174469142
Probability Amy is Brian's family = 0.3791735849751334
```

现在，观察体育主题为热门主题（sports.hot.observe(true)）并运行相同的查询。这将打印如下结果：

```
Probability Amy's interest is politics = 0.9609178093049061
Probability post 2's topic is sports = 0.3729396845525878
Probability post 3's topic is sports = 0.0900555124038995
Probability Amy is Brian's family = 0.33670840928905443
```

注意，帖子 2 和帖子 3 的主题是体育的概率已经增大。这可能令人吃惊，因为在 Post 类模型中，主题的热度没有成为主题选择的一部分。但是这是有原因的：您之前观察到帖子 1 的主题是政治。Amy 发布了帖子 1，所以她的兴趣更可能是政治。您还知道 Brian 发布了帖子 2 且 Amy 对此发表了评论。因为人们更倾向于评论与其兴趣匹配的主题，除非这些主题是热门话题，所以您相信帖子 2 的主题很可能是政治。但是，现在您知道体育是热门主题，因此，帖子 2 的主题是体育的可能性很大，Amy 发表评论是因为体育是热门主题，而非她对体育感兴趣。这就是帖子 2 的主题为体育的概率增大的原因。帖子 3 的情况与此类似。

现在，我们来思考 Amy 为 Brian 家人的概率为什么下降。您知道 Amy 最可能感兴

趣的是政治。帖子 2 有两种可能性：要么它是关于政治的，这种情况下不需要解释 Amy 发表评论的原因，要么是关于体育的，这种情况需要解释。两种可能的解释是：Amy 是 Brian 的家人，或者体育是热门主题。在观察到体育是热门主题之前，您对 Amy 是 Brian 家人的置信度大约有 38%。观察到体育是热门主题提供了另一个解释，因此降低了 Amy 是 Brian 家人的概率。这是您在第 5 章关于贝叶斯网络的论述中看到的诱导依赖性的一个例子。“Amy 是 Brian 的家人”和体育主题的热度原先是独立的，没有理由认为两者应该相互联系。但是在观察到 Amy 很有可能对政治感兴趣且对 Brian 的帖子发表评论之后，两者就产生了联系。这是**“解释消除”**推理模式的经典例子，其中有相同观测值的两种备选解释，观察到一种解释会降低另一种解释的可能性。

7.3.3　具有类型不确定性的打印机模型

您已经了解了如何实现具有关系不确定性的模型，下面我们用打印机类型未知的模型研究类型不确定性。这里的原理和上面类似：使用元素集合和引用。创建多个打印机实例，每个用于一个可能的类型，并创建一个取值为这些打印机实例之一的元素，以表示未知打印机。

这个例子有两个前一小节未见的新变化。首先，您将陈述证据，并查询身份未知的元素。所以在查询和观察中使用 get，这很简单。

第二个变化是，您没有从根本上改变 PrintExperience 的原始定义，该定义中以 Printer 为一个参数。您没有具体的打印机，而是用一个 Element[Printer]表示未知打印机，您必须创建一个 PrintExperience 并访问其属性。您将使用 Figaro 的 Apply 实现这一点。在得到由 Element[Printer]表示的未知打印机之后，使用 Apply 创建一个 Element[PrintExperience]，其中的 PrintExperience 基于任意打印机。

实际上，您将看到模型的变化很小，只有区区五行。更多的变化出现在模型实例化和查询的方法，但是那正是您打算修改的地方。下面是模型的 5 处变化。

1．object PrinterProblemTypeUncertainty extends ElementCollection {

我们将整个模型放在一个元素集合中。

2．abstract class Printer extends ElementCollection {

打印机是一个元素集合。

3．val powerButtonOn = Flip(0.95)("power button on", this)

powerButtonOn 是您将要查询的属性，我们为其命名，并将其放在所属的打印机元素集合中。

4．class PrintExperience(printer: Printer, software: Software, network: Network, user: User) extends ElementCollection {

打印机体验也是一个元素集合。

5．val summary = Apply(…)("summary", this)

您将观察关于打印机摘要的证据，所以为其命名，并将其放在打印机体验元素集合中。

变化就是这些。现在您可以描述一个特定的情况。这就是类型不确定性出现的地方。您将按照如下的方式推进。首先，定义 myPrinter 为不同类型打印机中的一次随机选择，为 myPrinter 命名，并将其放在整个模型的元素集合中：

```
val myPrinter =
  Select(0.3 -> new LaserPrinter,
         0.7 -> new InkjetPrinter)("my printer", this)
```

接下来，和以前一样创建 Software、Network 和 User 实例：

```
val mySoftware = new Software
val myNetwork = new Network
val me = new User
```

最后，使用 Apply 和现有的 PrintExperience 类创建 myExperience，以特定的打印机、软件、网络和用户作为参数。同样，为 myExperience 命名并将其放在最高级的元素集合中：

```
val myExperience =
  Apply(myPrinter,(p: Printer) =>
    new PrintExperience(p, mySoftware, myNetwork, me))("print experience",
                                                         this)
```

现在，您已经为陈述证据和提出查询做好了准备。您将观察有关打印体验摘要的证据。但是 myExperience 是一个 Element[PrintExperience]，如何得到这个摘要呢？当然是使用引用！您为摘要元素命名并将其放在 PrintExperience 元素集合中，并为 myExperience 命名，将其放在最高级的元素集合中，以便完成引用。代码如下：

```
val summary = get[Symbol]("print experience.summary")
summary.observe('none)
```

根据“无打印输出”的证据，您将查询两件事。一是打印机电源按钮是否开启。同样，myPrinter 是一个元素，所以不能直接查询，但是可以使用引用：

```
val powerButtonOn = get[Boolean]("my printer.power button on")
```

第二个查询更为有趣。想象您身处服务台，不知道用户的打印机类型。在用户开始告诉您有关打印体验的事情时，您可能开始臆测打印机的类型。您可以查询模型，根据证据得出打印机的类型。使用 Scala 的运行时类型检查方法 isInstanceOf 可以实现这一操作。确切地说，可以用如下代码定义一个元素，该元素在 myPrinter 是激光打印机时取值为 true：

```
val isLaser =
  Apply(myPrinter, (p: Printer) => p.isInstanceOf[LaserPrinter])
```

现在，您可以进行推理，回答查询，获得如下结果：

```
After observing no print result:
Probability printer power button is on = 0.8868215502107177
Probability printer is a laser printer = 0.23800361000850662
```

打印机为激光打印机的先验概率是 0.3。现在您已经观察到一次不好的打印体验，这个概率已经降低。根据我们的模型，激光打印机比喷墨打印机更可靠，所以糟糕的打印体验更可能发生在喷墨打印机上。

我希望您能了解，建立对象及其关系的模型（包括关系及类型不确定性），能够描述有趣和丰富的情况，进行复杂的推理。和前一章中描述的集合相结合，本章的面向对象和关系建模范式使您得到使用常规编程中十分熟悉的技术创建概率模型的全部技能。下一章描述如何建立关于重要特例的模型，并对其进行推理——随时间变化的情况的动态模型。

7.4 小结

- 面向对象和关系编程范式使您能够构造复杂的模型并高效地重用模型组件。
- 在这些范式中，您用属性和概率依赖性、函数形式及数值参数规定类模型。
- 在实例级别，这些模型定义了所有实例的所有属性上的概率分布；您可以陈述这些属性的相关证据，并对其进行查询。
- Figaro 元素集合和引用提供了访问编译时身份未知的元素的机制。
- 为了表现关系或者类型不确定性，创建值为未知元素集合的元素，并使用引用访问元素集合的属性。

7.5 练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．您的公司有 5 个部门：研发、生产、销售、人力资源和财务。构建一个面向对象概率模型，捕捉这些部门之间的影响。使用该模型，根据部门状态查询公司的健康状况。

2．您将建立电影受欢迎程度的模型。每部电影有多位演员，每位演员出演多部电影。每位演员有一个表示受欢迎程度的变量。电影的受欢迎程度取决于演员受到喜爱的程度。根据电影、演员及受欢迎程度的数据集，预测指定演员出演的新电影的受欢迎程度。

3．在您的学校中，学生可以选择不同学科的课程。同一位教师可能教授多个学科，同一个学科可能由多位教师教授。学生在某一课程中的成绩级别取决于 3 个因素：学生的能力、学科的难度和教师的质量。编写一个关系概率模型以表现这种情况。使用由学生、选择的课程、他们的教师以及成绩级别组成的数据集，推断目标学生的能力、目标学科的难度以及目标教师的质量。

4．继续练习 3，您是一位学生，正在规划下一学期的课程。不幸的是，教师还没有公布，但是您可以假设某个学科的教师将是之前的任课教师之一。使用关系不确定性表现这种情况，预测目标学科的成绩级别。

5．考虑一个车辆监控应用。存在多种车辆，如卡车、轿车和摩托车。每种车辆都有尺寸和颜色等属性。每种车辆的变量分布不同。您有一部照相机可以拍摄车辆的照片，有一个图像分析算法能够估计车辆的尺寸和颜色。估计的大小和颜色取决于真实的尺寸和颜色，但不一定完全相等。使用具有类型不确定性的关系模型表现这种情况。利用该模型根据估算的尺寸和颜色推断给定车辆的类型。

第 8 章　动态系统建模

本章介绍如下内容：

- 创建动态系统的概率模型
- 不同类动态模型的使用，包括马尔科夫链、隐含马尔科夫模型和动态贝叶斯网络
- 使用概率模型创建新型动态模型，如具有时变结构的模型
- 动态系统的持续监控

在过去几章中，您已经学习了许多关于使用概率编程构建概率模型的知识。现在，您已经掌握了许多技术，包括依赖性建模、函数、集合和面向对象建模。本章在这些技术基础上建立一种特别重要的系统模型：**状态随时间而改变的动态系统**。

在 8.1 节中介绍动态概率模型的一般概念之后，8.2 节通过一系列示例增强对概念的理解，从最简单的时间序列开始，到系统状态结构随时间变化的系统为止。首先，本章假定动态系统运行的时长固定，但是 8.3 节放松了这一假设，可以为无限运行的系统建模。这需要一个新的 Figaro 概念——**宇宙**。您将了解如何使用 Figaro 建立持续系统模型并进行推理。

8.1 动态概率模型

在第 1 章中，您研究了推理足球赛中角球的问题。利用概率推理系统可以回答 3 类查询：

- 根据给定的因素（如风力、中锋身高等）预测角球的结果。
- 推断可能导致观察结果的属性，如守门员的技能水平。
- 使用一次角球的结果推断可能影响第二次角球结果的属性，然后相应地预测第二次角球的结果。

在本章中，您不再为作为孤立事件的单次角球建模，而是将整场足球赛作为相互联系的一序列事件建模。足球赛是**动态系统**的一个例子。这意味着：

- 足球赛的每个时间点都有一个**状态**。这个状态可能包含比分、控球方和每个球队的信心。
- 任何时点的状态**取决于更早的状态**。例如，在足球赛中，任何时点的比分取决于之前的比分和是否刚刚有进球；球权取决于之前的球权，因为控球一方有机会保持控球；球队的信心也取决于之前的信心，因为信心通常不会突然地大幅度波动。

结合这两点就可以得出如下定义：**动态系统**是每个时间点有一个状态，不同时点的状态相互关联的系统。

动态系统的例子很多。天气明显是一个动态系统，因为今天的天气依赖于前几天的天气。类似地，公司绩效也是一个动态系统；其状态由多个数量组成，如收入和利润，一段时间内这些数量的值相互联系。第三个例子是公路上的交通状况；状态可能是每个车道上的车辆数量，很明显，某个时点的汽车数量与较早时点的数量相关。

您将研究动态系统概率模型的创建。这种模型称作**动态概率模型**。在动态概率模型中，状态由随机变量表示。例如，对于一家公司的绩效，您可能有表示任何时点收入水平及利润的变量。这些变量称作**状态变量**。概率模型定义不同时点状态变量值之间的概率依赖性。

动态概率模型的使用与常规静态概率模型的使用方式相对应。

- **考虑当前状态和一段时间的状态之间的依赖性，预测未来时点的系统状态**。例如，您可以考虑当前比分和球队的信心，预测足球赛的最终比分。
- **推断当前状态的过去根源**。例如，如果您的球队在比赛中失利，可以尝试确定比赛中的哪些决策导致这样糟糕的结果。
- **根据一段时间内得到的观测值，随时监控系统状态**。例如，您可以根据球场上发生的情况，持续估算两个球队的信心和质量。然后，您可以使用这些估算预测剩下的比赛时间内发生的情况。

上述 3 种能力对足球队的经理极其有用。实际上，动态系统十分重要和普遍，因此为动态概率模型已经开发了整组技术。概率编程系统可以表现用于动态概率模型的许多已有框架，如下一节介绍的隐含马尔科夫模型和动态贝叶斯网络，而且更进一步地包含了丰富数据结构和控制流等特性。好了，不再多言，让我们来研究一些此类框架，以及使用概率编程表达它们的方法吧。

8.2 动态模型类型

本节介绍各种不同的动态模型。您将从最基本的模型（马尔科夫链）开始，然后考虑广为使用的扩展模型——隐含马尔科夫模型。此后，您将研究动态贝叶斯网络，正如其名，这是贝叶斯网络在动态模型上的扩展。所有这些都是概率编程之前就出现的标准框架。它们都假定模型的结构在每个时点上都相同。概率编程使您可以超越这些框架，描述结构随时变化的模型，您将在 8.2.4 小节看到这种模型。

8.2.1 马尔科夫链

动态系统是包含随时变化状态的系统，不同时间的状态相互关联。马尔科夫链是最简单的动态系统，有两方面的特性：首先，状态由单一变量组成。其次，每个时点的状态变量概率依赖于前一时点的变量，但不是之前的任意状态变量。

图 8-1 展示了马尔科夫链用于足球赛的一个例子。在这个模型中，状态由单一变量组成，表示比赛中的某个特定时点哪个球队控球。为了方便起见，您可以说时点表示比赛中的每一分钟，但是它们可能是比赛的任何一部分。

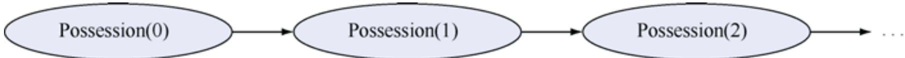


图 8-1 一个马尔科夫链。某一时点的状态由单一变量组成，该变量取决于前一时间的变量。Possession(1)直接取决于 Possession(0)，Possession(2)直接取决于 Possession(1)，依此类推，Possession(n)直接取决于 Possession(n-1)

观察图 8-1，您会看到箭头顺序地从一个状态指向下一状态，这说明：

- Possession(1)直接取决于 Possession(0)。
- Possession(2)直接取决于 Possession(1)，但不取决于 Possession(0)。
- Possession(0) 对 Possession(2)的任何影响都通过 Possession(1)中介。

另一种说法是，Possession(2)在 Possession(1)给定的情况下**条件独立**于 Possession(0)。对于未来的时点也是如此：**在前一时点的控球状态给定的情况下，任何时点的控球状**

态条件独立于任何更早时点的控球状态。这一陈述是称为**马尔科夫假设**的属性的一个例子。

马尔科夫假设：如果任何时点的状态仅依赖于前一个状态；任何时点的状态在前一个状态给定的情况下条件独立于之前的所有状态，则动态概率模型满足**马尔科夫假设**。

指定一个马尔科夫链

创建马尔科夫链需要做几件事。

1．决定状态变量的值。这包括您认为在任何时点相关的所有变量值。在此通常应该谨慎，不要包含不必要的值。您很快会看到，马尔科夫链表现形式的大小是状态变量值数量的二次方程式，所以要避免使其变得过大。例如，您可能有一个用于足球赛的马尔科夫链，其中的状态变量代表分差。原则上，分差可能很大——如果任一分钟都可能进球，最终的比分可能是 90∶0！但是在实践中，大分差很少见，而且，5～15 的分差也不合逻辑，因为在职业足球赛中，一个球队落后 5 球的情况很少。所以您可以将分差限制在−5～+5 的范围内。

2．完成如下工作之一。

- 为马尔科夫链指定一个初始值，也就是第一个状态变量的值。例如，足球赛开始时的分差为 0。
- 指定初始值上的一个分布。如果您不知道精确值，这就是必需的。例如，球赛开始时的球权依靠掷硬币决定，所以每个球队控球的概率为 0.5。

3．指定迁移模型。迁移模型定义一个时点的状态依赖于前一时点状态的形式。在图 8-1 中，迁移模型指定了在给定 Possession(0)的情况下 Possession(1)的概率、给定 Possession(1)的情况下 Possession(2)的概率……用符号表示，该迁移模型指定 P(Possession(t) | Possession($t-1$))　（$t \geqslant 1$）。

在第 3 步中，我假设所有不同时点的迁移概率都相同。这个假设不是对所有马尔科夫链都严格需要的，但是很实用，因此在大部分应用中都使用。利用这个假设，可以一个简单的循环定义马尔科夫链。

您将会发现，动态模型的处理代价可能很高。动态模型表现和推理的复杂度主要取决于状态的数量。在马尔科夫链中，迁移模型中的参数数量是状态变量值数量的二次方程式。在我们的例子中，迁移模型指定 P(Possession(t) | Possession($t-1$))。这是一个条件概率分布，可以由指定 Possession($t-1$) 和 Possession(t)的所有值概率的表格定义。表格中的条目数量是 Possession 变量可能值数量的平方。

在 Figaro 中使用马尔科夫链

如果您预先知道时间步的总数，在 Figaro 中描述一个马尔科夫链很容易。如果不知道时间步的总数，系统可能无限运行，就需要更高级的技术，这些技术在 8.3 节中介绍。但是，如果您知道时间步总数，可以编写一个简单的循环。

下面是定义图 8-1 中的马尔科夫链的代码：

程序清单 8-1　马尔科夫链规格

```
val length = 90                                    ← 链的长度

val ourPossession: Array[Element[Boolean]] =       状态变量数组，每
  Array.fill(length)(Constant(false))              个时间步一个

ourPossession(0) = Flip(0.5)                       ← 为序列的初始状态设置分布
for { minute <- 1 until length } {
  ourPossession(minute) =
    If(ourPossession(minute - 1), Flip(0.6), Flip(0.3))   迁移模型根据序列中的前一个状态，定义了每个状态变量的分布
}
```

首先将序列的长度设置为 90 分钟。然后，创建一个数组，对于 0～89 的每个时点有一个元素，该元素是一个布尔状态变量，表示您的球队是否控球。Scala 方法 Array.fill 创建一个数组，长度由第一个参数（length）给定，每个元素的值初始化为 Constant(false)元素。这个初始值并不重要，因为您很快就将覆盖它。

接下来，定义球队在时点 0 是否控球的概率分布。您已经选择用 Flip(0.5)定义，所以在初始时间点控球的概率为 0.5。

接下来是从时点 1 到时点 89 的一个循环。在每个时点，根据前一时点是否控球，定义该时点是否控球。确切地说，如果您在前一时点控球，继续控球的概率为 0.6，如果前一时点没有控球，此时夺得球权的概率为 0.3。

您可以查询这个马尔科夫模型，根据任何时点的观测值，得出任意时点状态变量的概率分布。例如，假定您打算查询时间步 5 时控球的概率。可以调用如下函数，询问观察到任何证据之前的概率：

```
VariableElimination.probability(ourPossession(5), true)
```

这将返回时间步 5 时您的球队控球的先验概率——0.428745。
然后，可以用如下代码观察时间步 4 时您的球队控球的情况：

```
ourPossession(4).observe(true)
```

相同的查询此时返回 0.6。可以看到，在得知前一时间步您的球队控球的情况下，这一概率有所增大。实际上，这个概率就是在一个时间步中保持控球的概率。

如果您观察到在时间步 3 时，球队也控球，那么会发生什么呢？查询该模型，仍然返回您在第 5 步控球的概率为 0.6。新的观测值没有改变这一概率。这是因为马尔科夫假设：在时间步 4 是否控球已经确定的情况下，在时间步 5 是否控球独立于时间步 3 是否控球。

您也可以观察时间步 6 拥有球权的情况。这说明不仅可以通过马尔科夫链向前预测未来，也可以从未来的观测值中向后推导出之前的状态。在添加这一观测值之后，您在

时间步 5 控球的概率上升到 0.75。

最后，您观察到在时间步 7 时球队也控球。答案仍然是 0.75（在舍入误差范围内）。这是马尔科夫假设的另一个实例。在第 6 分钟的情况已知之后，在第 7 分钟是否控球的证据不会为第 5 分钟的查询增添任何新信息。

马尔科夫链是动态概率模型的最简单情况，但是为更强大的模型提供了基础，下面几个小节详细说明这些模型。

8.2.2 隐含马尔科夫模型

隐含马尔科夫模型（HMM）是马尔科夫链的扩展，每个时间点存在两个变量，一个代表某种“隐含”状态，另一个表示某个观测值。图 8-2 展示了 HMM 的一个例子。隐含状态表示您的球队在每个时点是否自信。您从未真正知道球队是否自信，必须从场地上发生的情况推断。这就是它成为隐含状态的原因。可以直接观察到的观测值是“是否控球”。

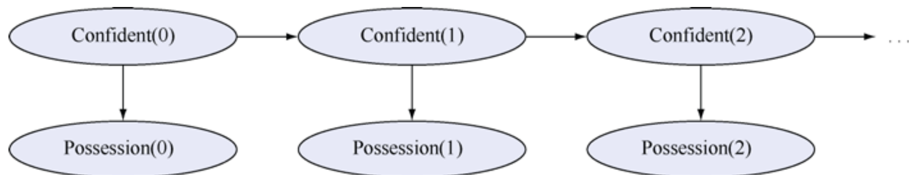


图 8-2　一个隐含马尔科夫模型。Confident（自信）是一个隐含状态变量，Possession（控球）表示一个观测值。隐含状态变量形成一个马尔科夫链，观测值只依赖于该时点的隐含状态

从图 8-2，可以看出 HMM 满足两个假设。

- 隐含状态形成了一个满足马尔科夫假设的马尔科夫链。
- 某个时点的观测值仅依赖于当时的隐含状态。在某个时点的隐含状态给定时，观测值独立于之前的所有隐含状态和观测值。

这还意味着，**在前一个隐含状态给定的情况下**，特定时点的隐含状态独立于所有之前的观测值。但是有一个要点必须说明。按照假设，前一个隐含状态是“隐含”的，所以您通常不能够确知。如果不知道前一个隐含状态，当前隐含状态不独立于之前的观测值。例如，在图 8-2 中，如果没有观察到 Confident(0) 或 Confident(1)，Possession(0)不独立于 Confident(2)。

上述要点对于理解 HMM 的实用性和强大表现力是必不可少的。前面的两个假设很重要，它们使 HMM 的表现形式更加紧凑，推理更加高效。与此同时，它们不会妨碍您使用**整个**观测值序列推断隐含状态。

用 Figaro 描述一个 HMM

知道如何描述马尔科夫链之后，描述 HMM 也不是太复杂。您需要做以下的工作。

1．定义一组隐含状态变量值。在我们的例子中，Confident 可能是一个布尔变量。因为隐含状态变量组成一个马尔科夫链，表现形式的大小是隐含状态变量值数量的二次方程式。

2．定义一组观测变量值。在我们的例子中，Possession 也可能是一个布尔变量，表示特定时点我们的球队是否控球。因为观测变量依赖于隐含状态而不是前一个观测值，表现形式的大小将与隐含状态值数量和观测值数量的乘积成正比。

3．定义初始隐含状态上的一个概率分布。这被称作**初始模型**。在我们的例子中，这将指定 P(Confident(0))。

4．定义隐含状态变量的迁移模型，表示给定前一时点情况下，状态变量在某个时点的条件分布。在我们的例子中，这将指定 P(Confident(*t*) | Confident(*t* – 1))。

5．定义观测模型，指定在该时点隐含状态变量给定的情况下，任何时点观测变量上的条件概率分布。在我们的例子中，这将指定 P(Possession(*t*) | Confident(*t*))。

下面的程序清单说明了上述步骤在代码中的实现。

程序清单 8-2 HMM 规格

```
val length = 90

val confident: Array[Element[Boolean]] =
  Array.fill(length)(Constant(false))
val ourPossession: Array[Element[Boolean]] =
  Array.fill(length)(Constant(false))


confident(0) = Flip(0.4)

for { minute <- 1 until length } {
  confident(minute) = If(confident(minute - 1), Flip(0.6), Flip(0.3))
} //

for { minute <- 0 until length } {
  ourPossession(minute) = If(confident(minute), Flip(0.7), Flip(0.3))
} //
```

隐含状态变量数组，每个时间步一个

观测变量数组，每个隐含状态变量一个

设置序列初始隐含状态的分布

根据序列中前一个隐含状态，定义每个隐含状态变量分布的迁移模型

在对应的隐含状态变量给定的情况下定义每个观测变量分布的观测值模型

当您从观测值推断某个时点的隐含状态时，可以考虑 3 类观测值：当前观测值、之前的观测值和未来的观测值。**未来的观测值**听起来很荒谬，怎么可能观察到未来？我的意思是可以根据从某个时点开始到现在为止接收到的观测值，推导较早时点的隐含状态。从较早的时间点看来，这些就是未来的观测值。

如前所述，推断隐含状态需要考虑所有观测值，而不仅仅是当前和最邻近的观测值。这可以由我们的程序中的如下查询说明。

程序清单 8-3　HMM 查询

```
println("Probability we are confident at time step 2")
println("Prior probability: " +
  VariableElimination.probability(confident(2), true))
ourPossession(2).observe(true)
println("After observing current possession at time step 2: " +
     VariableElimination.probability(confident(2), true))
ourPossession(1).observe(true)
println("After observing previous possession at time step 1: " +
  VariableElimination.probability(confident(2), true))
ourPossession(0).observe(true)
println("After observing previous possession at time step 0: " +
  VariableElimination.probability(confident(2), true))
ourPossession(3).observe(true)
println("After observing future possession at time step 3: " +
  VariableElimination.probability(confident(2), true))
ourPossession(4).observe(true)
println("After observing future possession at time step 4: " +
  VariableElimination.probability(confident(2), true))
```

运行上述程序打印如下结果：

```
Probability we are confident at time step 2
Prior probability: 0.42600000000000005
After observing current possession at time step 2: 0.6339285714285714
After observing previous possession at time step 1: 0.6902173913043478
After observing previous possession at time step 0: 0.7046460176991151
After observing future possession at time step 3: 0.7541436464088398
After observing future possession at time step 4: 0.7663786503335885
```

可以看到，每个观测值都增大了对时间步 2 时“球队自信”的置信度。

8.2.3　动态贝叶斯网络

尽管 HMM 在从一系列观测值推断隐含状态这方面很强大，但是它们仍然是在每个时点只使用两个变量的简单表现形式。一般来说，您感兴趣的变量可能很多，它们可能以不同方式相互依赖。**动态贝叶斯网络**（DBN）是 HMM 服务于这一需求的推广。它们采用和 HMM 相同的原理——建立随时间变化的一系列变量的模型，但是可以有许多个变量，各个变量之间可以有趣的方式相互依赖。

图 8-3 展示了一个用于足球赛的 DBN 示例。顾名思义，DBN 类似于贝叶斯网络，包含的内容也相似。

DBN 有如下内容。

1．每个时点的一组状态变量。在图 8-3 中有如下变量：

图 8-3　一个动态贝叶斯网络。每个时点存在多个变量。某个时点的变量可能依赖于该时点的其他变量，或者前一时点的变量

- Winning，表示目前领先的球队。
- Confident，表示您的球队是否自信。
- Possession，表示您的球队是否控球。
- Goal，表示在那个时点是否进球。
- ScoreDiff，表示两个球队之间的分差。

2．每个变量有一组变量值。在我们的例子中：

- Winning 可能为“us”（我方）、“them”（对方）或者“none”（无）。
- Confident 是一个布尔变量。
- Possession 是布尔变量。
- Goal 是布尔变量。
- ScoreDiff 是从-5 到 5 的整数。

3．一个迁移模型，包括：

- 每个变量的一组父变量。这些父变量可以是同一时间步的其他变量，或者前一时间步的变量。父变量的唯一限制是（1）边不能穿过超过一个时间步，（2）和贝叶斯网络中一样，边不能形成有向环。在我们的例子中，有如下依赖性。
  —— 您在某个时点是否领先（Winning）依赖于前一时点的 ScoreDiff。
  —— 您在某个时点是否自信（Confident）依赖于谁目前领先（Winning）和之前是否自信。
  —— 是否控球（Possession）依赖于您目前是否自信。
  —— 任一方是否进球（Goal）依赖于您是否控球和您的自信。

—— 新的 ScoreDiff 由前一个 ScoreDiff、是否有一个进球（Goal）和哪一方进球（由您是否控球决定）决定。

■ 对于每个变量，有一个条件概率分布（CPD）规定每个父变量值上的概率分布。在我们的例子中，CPD 如下。

—— Winning 显然由之前的 ScoreDiff 决定。

—— Confident 的 CPD 规定，如果您领先（Winning）且之前是自信的，那么您就更有可能自信。

—— Possession 的 CPD 规定，如果您自信，就更有可能控球。

—— Goal 的 CPD 规定，如果球队控球且自信，就更有可能进球。这意味着，如果您控球且自信，或者您没有控球且不自信，就更有可能出现进球。

—— 新的 ScoreDiff 是前一个 ScoreDiff、是否进球和是否控球（这决定了您或者对方是否进球）的确定性函数。

4．指定初始时间点变量概率分布的初始模型。一般来说，这是初始变量上的一个常规贝叶斯网络。但是您只需要对下一时点有影响的变量上的分布，就可以推动 DBN 运行。在我们的 DBN 中，这些变量是 Confident(0) 和 ScoreDiff(0)。

从上述列表中可以看到，DBN 规格的主要内容在迁移模型中。一般来说，它不像图 8-3 那样展示变量的时间序列（图中展示了时间点 0、1、2 等的变量），而是展示时间步和前一时间步之间的关系。这种关系称作**两个时间步的贝叶斯网络**，简称 2TBN。图 8-4 展示了我们的示例中的 2TBN。

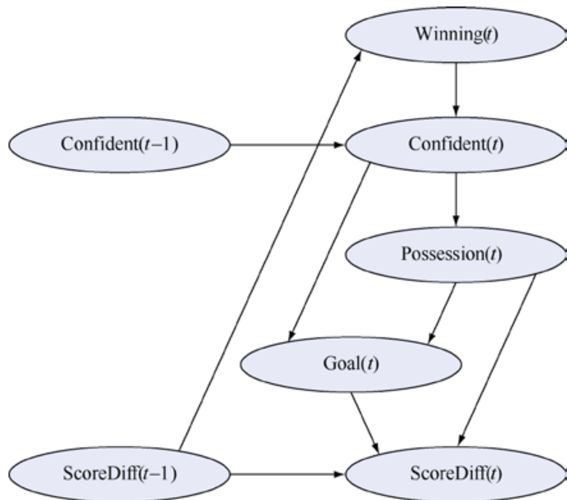


图 8-4 一个包含两个时间步的贝叶斯网络（2TBN）。这个 2TBN 展示了时间步 t 的变量和它们与时间步 t-1 的依赖性

从图中可以看出，与常规贝叶斯网络类似，2TBN 中的变量是时间步 t 和 t-1 的状态变量。

唯一的区别是，时间步 t-1 的变量没有父变量，也没有 CPD；只定义了时间步 t 的变量依赖性和分布。2TBN 是之前描述的 DBN 迁移模型的一种编码，通常也是 DBN 的描述手段。

在 Figaro 中描述 DBN

您已经知道如何在 Figaro 中描述常规的贝叶斯网络和 HMM。因为 DBN 是这两种框架的组合，所以您已经有了描述它的必要技术。

程序清单 8-4 图 8-3 中 DBN 的 Figaro 实现

```
val length = 91
val winning: Array[Element[String]] = Array.fill(length)(Constant(""))
val confident: Array[Element[Boolean]] =
  Array.fill(length)(Constant(false))
val ourPossession: Array[Element[Boolean]] =
  Array.fill(length)(Constant(false))
val goal: Array[Element[Boolean]] =
  Array.fill(length)(Constant(false))
val scoreDifferential: Array[Element[Int]] =
  Array.fill(length)(Constant(0))
```

在每个时间点创建 5 个状态变量的数组

```
confident(0) = Flip(0.4)
scoreDifferential(0) = Constant(0)
```

创建 Figaro 元素，表示 Confident 和 scoreDifferential 的初始值

```
for { minute <- 1 until length } {
  winning(minute) =
    Apply(scoreDifferential(minute - 1), (i: Int) =>
            if (i > 0) "us" else if (i < 0) "them" else "none")
  confident(minute) =
    CPD(confident(minute - 1), winning(minute),
        (true, "us") -> Flip(0.9),
        (true, "none") -> Flip(0.7),
        (true, "them") -> Flip(0.5),
        (false, "us") -> Flip(0.5),
        (false, "none") -> Flip(0.3),
        (false, "them") -> Flip(0.1))
  ourPossession(minute) = If(confident(minute), Flip(0.7), Flip(0.3))
  goal(minute) =
    CPD(ourPossession(minute), confident(minute),
        (true, true) -> Flip(0.04),
        (true, false) -> Flip(0.01),
        (false, true) -> Flip(0.045),
        (false, false) -> Flip(0.02))
  scoreDifferential(minute) =
    If(goal(minute),
      Apply(ourPossession(minute), scoreDifferential(minute - 1),
            (poss: Boolean, diff: Int) =>
              if (poss) (diff + 1).min(5) else (diff - 1).max(-5)),
      scoreDifferential(minute - 1))
}
```

该循环定义每个时间点的迁移。迁移模型规定 5 个状态变量如何依赖于之前的自信状态和分差（scoreDifferential）。每个变量也依赖于循环中较早的变量。例如，confident 依赖于同一时点的 winning

您可以使用 DBN 模型进行各种查询。例如，您可以根据当前状态预测比赛的未来，或者推断导致当前结果的前一状态。对于足球教练（或者赌球者）来说，实用的用法之一是预测谁能赢得比赛。如果在前一个程序中查询最后的 scoreDifferential 变量，就可以得出赢下比赛的概率大约为 0.4。但是如果观察到您的球队在第 4 分钟获得一个进球，该概率就会上升到大约 0.73。

8.2.4　结构随时间改变的模型

迄今为止您所看到的 3 类模型——马尔科夫链、隐含马尔科夫模型和动态贝叶斯网络——是标准素材。它们包含了一组固定结构的固定变量。概率编程为您提供处理更高级模型的能力，这种模型中的状态结构可能随时间而变化。

例如，想象一位饭店老板想要知道某个晚上饭店的客满情况；这可能帮助她做出各种决定，比如制作多少食物、鼓励客户逗留还是快速离开，等等。饭店是涉及客户进店、享用晚餐和离开的动态系统。该系统的状态可能由目前店中客人数和他们已经停留的时间以及等座客人数量等组成。因为客人的数量随时间而变化，状态的结构也随之变化。而且，您无法预先知道任何时点的结构；在任何给定的时点，您必须考虑许多种可能的结构。

使用概率编程建立可变结构系统的模型

用概率编程建立此类系统模型的最简单方法是创建一组状态变量，这些变量的类型是可变数据结构。在我们的例子中（如图 8-5 所示），可以有两个状态变量。

- Seated 代表目前在饭店中就坐的客人，以及他们在此停留的时间。其数据类型是整数的列表。列表的长度是当前就坐的客人数量，每个整数代表某位客人已经就坐的时间。
- Waiting 代表当前等座的客人数量，这是一个简单的整数。

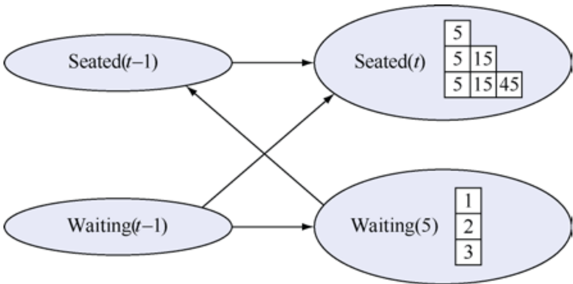


图 8-5　变量结构随时变化的动态模型。每个状态变量展示了 3 个可能值。Seated 变量的可能值是变长数组

用这些丰富数据类型的状态变量建立模型之后，该模型就像一个 DBN，可以用类似的

方法编码。但是，需要一些额外的技术以处理丰富数据类型的变量。我们逐步地建立模型。

首先，设置饭店的容量和运行模型的步骤数量。为了简单，假设饭店有 10 张同等大小的桌子，每组客人占用一张桌子，所以容量为 10。此外，假定您以 5 分钟为一个时间步进行推理，为期 1 个小时，所以步数为 12。

```
val numSteps = 12
val capacity = 10
```

对于前面描述的两个状态变量，各用一个数组表示不同时间的变量值，每个变量都有对应的类型。所以，每个 seated 变量是 Element[List[Int]]，waiting 则为 Element[Int]：

```
val seated: Array[Element [List[Int]]]=
  Array.fill(numSteps)(Constant(List()))
val waiting: Array[Element[Int]] = Array.fill(numSteps)(Constant(0))
```

本例假定饭店老板从晚上某个已知状态的时点开始，所以您将使用 Figaro 的 Constant 结构将状态变量的初始值设置为已知值。在初始时点，饭店已经客满，有人正在等位。

```
seated(0) = Constant(List(0, 5, 15, 15, 25, 30, 40, 60, 65, 75))
   waiting(0) = Constant(3)
```

表现迁移模型

下面进入有趣的部分了：迁移模型。您将分两个步骤编写该模型。首先，编写迁移模型的骨架，然后完善细节。下面是迁移模型的骨架。

程序清单 8-5 可变结构的动态模型——代码骨架

```
def transition(seated: List[Int], waiting: Int):
  (Element[(List[Int], Int)]) = {
  // details go here
  ^^(allSeated, newWaiting)
}

for { step <- 1 until numSteps } {
  val newState =
    Chain(seated(step - 1), waiting(step - 1),
     (l: List[Int], i: Int) => transition(l, i))

  seated(step) = newState._1
  waiting(step) = newState._2
}
```

定义一个迁移函数，以之前的状态变量为参数，返回新状态变量上的联合概率分布

由前几个状态变量上的分布生成新状态变量上的联合分布

从这个联合分布提取新的状态变量

仔细观察这段代码，因为它展示了一种通用的模式。您可能不知道**马尔科夫链**中的“链”和 Figaro 的链接有何相关。两者之间存在紧密的联系。目前我们还没有注意到这一点，但是，从一个时间步到下一个时间步的状态变量链接可以用 Figaro 的 Chain 实现。还记得吗？Chain 取得父变量上的一个概率分布、从父变量到子变量的 CPD，生成子变量上的一个概率分布。在马尔科夫链中，父变量是 possession(t – 1)之类的变量，而子变

量是 possession(t)之类的变量，CPD 则是迁移模型，在 possession(t – 1)给定的情况下提供 possession(t)上的概率分布。所以，要得到 possession(t)上的分布，可以编写如下代码：

```
Chain(possession(t – 1), transitionModel)
```

上述代码使用了类似的逻辑。您拥有时间 step-1 的状态，该状态由 seated(step – 1) 和 waiting(step – 1)组成。迁移函数取得 seated 和 waiting 的特定值，生成下一状态上的一个元素，该元素是一对 List[Int]（用于 seated）和 Int（用于 waiting）。使用 Chain，从当前状态获得下一状态上的元素。

下一个状态是一对 seated 和 waiting 值上的元素。它定义了新时间步上 seated 和 waiting 变量的联合分布。在迁移函数中，这个元素使用 Figaro 的^^构造程序构造，该程序从单独元素创建对组（或者更大的元组）上的元素。在这个迁移函数中，这些元素是 allSeated 和 newWaiting，您很快会看到它们是如何生成的。

最后，链接的结果是一对变量上的元素。您应该从这个配对中取出单独的 seated 和 waiting 元素，使用 Figaro 的_1 和 _2 提取操作可以实现。_1 取得一个配对上的元素，创建第一个成分上的元素，_2 的功能也类似。

下面是详细的迁移函数。

程序清单 8-6 可变结构的动态模型——迁移函数细节

```
def transition(seated: List[Int], waiting: Int):
  (Element[(List[Int], Int)]) = {
  val newTimes: List[Element[Int]] =
    for { time <- seated }
    yield Apply(Flip(time / 80.0),
                (b: Boolean) => if (b) -1 else time + 5)

  val newTimesListElem: Element[List[Int]] = Inject(newTimes:_*)

  val staying = Apply(newTimesListElem,
                      (l: List[Int]) => l.filter(_ >= 0))

  val arriving = Poisson(2)
  val totalWaiting = Apply(arriving, (i: Int) => i + waiting)

  val placesOccupied =
    Apply(staying, (l: List[Int]) => l.length.min(capacity))
  val placesAvailable =
    Apply(placesOccupied, (i: Int) => capacity - i)
  val numNewlySeated =
    Apply(totalWaiting, placesAvailable,
          (tw: Int, pa: Int) => tw.min(pa))
```

确定每位客人在饭桌上就坐的新时间。−1 表示他们正要离开，发生的概率为 time/80

newTimes 是一个 Element [Int]的列表。您需要将其转换为 Element [List [Int]]，这通过 Inject 实现

删除所有就坐时间小于 0 的客人，确定将要离开的客人数量

确定到达饭店的人数，得出等待人数

确定等位的客人中可以就坐的人数

```
    val newlySeated =
      Apply(numNewlySeated, (i: Int) => List.fill(i)(0))
    val allSeated =
      Apply(newlySeated, staying,
            (l1: List[Int], l2: List[Int]) => l1 ::: l2)

    val newWaiting = Apply(totalWaiting, numNewlySeated,
                           (tw: Int, ns: Int) => tw - ns)

    ^^(allSeated, newWaiting)
  }
```

确定新的就坐客人列表

确定新的等位客人数量

返回在一对新等位客人列表和等待客人数上的元素

因为这段代码使用的大部分都是您已经了解的 Figaro 功能，所以我就不再详加解释。但是有一个功能值得注意：Figaro 的 Inject 元素可以将一个元素列表转换为一个列表元素。 这是什么意思呢？假定您有一个由 Element[Int]组成的列表。一组可能值为：第一个元素为 5，第二个为 10，第三个为 15。您可以将这些元素转换为包含数值 5、10 和 15 的列表。在这种情况下，Inject(l)将为 List(5, 10, 15)。

这有什么用呢？在我们的模型中，确定客人就坐的新时间以及客人是否将要离开等操作适用于单个元素。而其他操作（如确定停留的客人列表）则适合于将所有客人作为整体的列表。这时候需要列表上的元素，Inject 可以提供。

利用模型进行推理

利用这个模型进行推理没有太多值得说明的地方，它和您前面已经了解的几类动态概率模型很相似。同样，您可以使用该模型预测未来，或者推断观察到的事件的根源。8.3 节介绍使用相同的模型随时监控饭店状态的方法。

一般来说，您可能不会使用变量消除法或者置信传播等因子分解算法。状态变量可能值的数量很大。在饭店的例子中，每个可能的就坐时间是 0 和 75 之间可以被 5 整除的数。（一旦就坐时间达到 75，客人在下一时间步肯定会离开）这就产生了 16 种可能值。10 个座位的就坐时间列表数量为 16^{10}。您还需要考虑客人数量少于 10 个的情况。这样造成的组合太多，无法一一列举。

因此，抽样算法工作得更好。本例使用重要性抽样预测下一小时的最后进入饭店等座的人数。下面是代码：

```
val alg = Importance(10000, waiting(numSteps - 1))
alg.start()
println(alg.probability(waiting(numSteps - 1), (i: Int) => i > 4))
```

运行这段代码产生 0.4693 左右的数值，这意味着这一小时的最后等位的人数大于 4 的概率大约为 47%。

现在，您已经看到了从简单到高级的各种动态概率模型。目前我所展示的所有程序都有一个局限性：必须预先定义时间步的数量。下一小节介绍如何突破这一限制。

8.3 建立永续系统的模型

本节的目标是定义可以持续任意时长的动态概率模型。这些模型可用于根据随时间推移积累的证据，监控持续运行的系统状态。实现这种模型的机制并不难，但是您需要了解一个新的 Figaro 概念——**宇宙**（universe）。

8.3.1 理解 Figaro 的宇宙概念

在第 7 章中，您学习了关于元素集合的知识。**元素集合**是一种数据结构，其中的元素按名访问。**宇宙**是一种特殊的元素集合，它还提供了对推理算法有用的服务，如内存管理和依赖性分析。因此，推理算法通常在宇宙上运行。

到目前为止，您对**宇宙**的概念还一无所知，大部分时候您没有必要考虑它。但是每个元素都属于某个宇宙。**始终有一个默认的宇宙**，除非另做指定，否则元素就处于这个默认宇宙中。

元素集合和宇宙的概念紧密相关。下面是需要记住的一些要点。

- 每个宇宙都是元素集合，但不是每个元素集合都是宇宙。
- 每个元素集合都与一个宇宙关联。如果元素集合本身就是宇宙，关联的宇宙就是它自己。
- 当您将元素放在一个元素集合中时，它的宇宙就是与该元素集合关联的宇宙。

您一定记得，可以通过提供可选的名称和元素集合参数，将一个元素放在元素集合中。例如，编写如下代码：

```
Flip(0.5)("coin", collection)
```

元素 Flip(0.5)被命名为 coin，放入由 collection 标识的元素集合中。因为宇宙是一个元素集合，您可以将宇宙作为第二个可选参数，将元素直接放入某个宇宙中，例如：

```
Flip(0.5)("coin", universe)
```

始终有一个当前的默认宇宙。如果创建元素时没有提供可选名称和元素集合参数，其元素集合和宇宙都将是默认宇宙。这个默认宇宙保存在 com.cra.figaro.language 包中的 Scala 变量 Universe.universe。

算法也在宇宙上运行。除非另外指定，这个宇宙就是当前默认宇宙。可以通过在参数列表中提供可选的附加参数指定不同的宇宙。例如，VariableElimination (target)在默认宇宙的指定目标创建一个变量消除算法。VariableElimination(target)(u2)在宇宙 u2 的给定目标上创建一个变量消除算法。VariableElimination.probability 等推理快捷方式始终在默认宇宙上运行。

您可以两种方式得到一个新的宇宙。其中之一是调用如下语句：

```
new Universe
```

这将创建一个没有元素的新宇宙。另一种方法是调用：

```
Universe.createNew()
```

这条语句除了创建新宇宙之外，还将默认宇宙设为该新宇宙。这提供了从新宇宙中开始工作，将新元素放入该宇宙，让您的算法在这个宇宙上运行的方便途径。例如，您可以编写如下语句：

```
Universe.createNew()
val x = Beta(1, 2)
val y = Flip(x)
println(VariableElimination.probability(y, true))
```

元素 x 和 y 将放入新宇宙，变量消除算法将在新宇宙上运行。

这里介绍的关于宇宙的一切知识都适用于在交互式 Scala 解释程序中运行的 Figaro。您所创建的所有元素都会进入一个宇宙，除非另外指定，该宇宙就是默认宇宙。如果想要忘记之前的交互，从头开始，输入 Universe.createNew()。所有新元素将进入新创建的默认宇宙，所有旧元素将被忽略。

8.3.2　使用宇宙建立持续运行系统的模型

为了用 Figaro 表现没有时间限制的动态概率模型，创建一个用于每个时间步的宇宙，包含该时间步的所有变量。该模型通常分成两部分描述：

- 一个初始宇宙。
- 从一个宇宙进入下一个宇宙的函数。

这两部分定义了如下的动态概率模型（如图 8-6 所示）。

1．从时间点 0 的初始状态上的一个概率分布开始，这个概率分布由初始宇宙中的所有元素描述。

2．对初始宇宙应用函数，得到时间点 1 的宇宙。新宇宙中的元素定义了时间点 1 状态上的概率分布。

3．对时间点 1 的宇宙应用函数，得到时间点 2 的宇宙。新宇宙中的元素定义了时间点 2 状态上的一个概率分布。

4．根据需要继续。

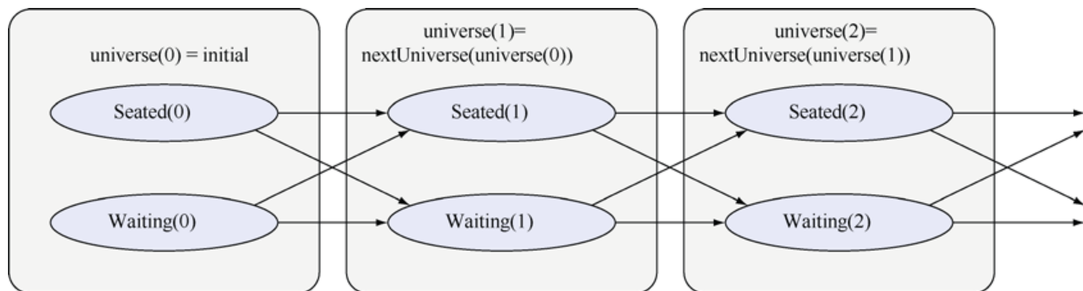


图 8-6　动态模型通过一系列宇宙推进。每个宇宙包含给定时点的状态变量。第一个宇宙由初始模型定义，后续的宇宙通过对前一个宇宙应用 nextUniverse 函数创建

在一个时间步中直接影响下一时间步中元素的任何元素都必须命名。这个名称使程序能够引用前一宇宙中的元素。在饭店的例子中，在一个时间步结束时等座的客人数影响下一时间步的状态。您必须将表示这一数量的元素命名为 waiting。

现在，迁移函数以前一个宇宙作为参数。假设这个宇宙包含在 Scala 变量 previous 中，您可以用代码 previous.get[Int]("waiting")获得前一个宇宙中名为 waiting 的元素。注意，您必须告诉 Scala 这个元素的值类型（Int），否则 get 方法将无法知道返回的元素类型。

您希望查询或观察的元素也需要命名。那样，就可以在每个时间步中一致地引用该元素。这段开场白之后，下面就是表现饭店持续模型的代码骨架。

程序清单 8-7　nextUniverse 函数实现

```
def transition(seated: List[Int], waiting: Int):
  (Element[(List[Int], Int, Int)]) = {
    // details are the same as before
    ^^(allSeated, newWaiting, arriving)
}

def nextUniverse(previous: Universe): Universe = {
  val next = Universe.createNew()
  val previousSeated = previous.get[List[Int]]("seated")
  val previousWaiting = previous.get[Int]("waiting")
  val state = Chain(previousSeated, previousWaiting, transition _)
```

迁移函数没有变化，只是在返回元素中加入了到达人数，因为您将对其进行观察

定义从上一宇宙到下一宇宙的函数

从前一个宇宙获得之前的状态变量

创建新宇宙，使其成为默认宇宙，并将其赋给一个 Scala 变量

使用 Chain 获得代表新状态的元素

```
  Apply(state, (s: (List[Int], Int, Int)) => s._1)("seated", next)
  Apply(state, (s: (List[Int], Int, Int)) => s._2)("waiting", next)
  Apply(state, (s: (List[Int], Int, Int)) => s._3)("arriving", next)
  next
}
```

返回下一个宇宙

从这个元素获得表示单独状态变量的元素，在下一个宇宙中分别命名

从这段代码可以看到，这和模型的限时版本没有太多不同。特别是，包含主要逻辑的迁移函数几乎没有变化。代码中最为明显的新特征是使用名称标识每个宇宙中代表状态变量的元素。您通过名称从前一个宇宙中获得这些元素，在放入新宇宙时为其命名。

这就是模型的表现方法。现在，我们来看看如何创建一个应用程序，以持续的方式监控系统状态。

8.3.3 运行一个监控应用

您的目标是从关于系统状态的初始信念开始，根据每个时间步接收到的证据，重复更新关于状态的信念。确切地说，您执行的是如下过程（参见图 8-7）。

1. 从时间点 0 系统状态上的一个分布开始。
2. 加入时间点 1 接收到的观测值，产生时间点 1 系统状态上的一个分布。
3. 加入时间点 2 接收到的观测值，产生时间点 2 系统状态上的一个分布。
4. 根据需要重复。

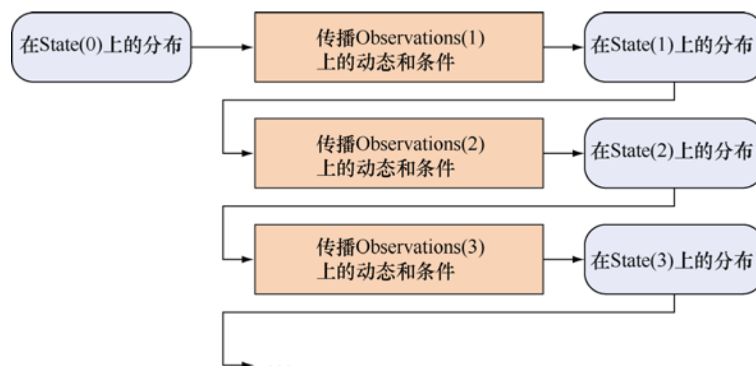


图 8-7 过滤过程。Figaro 维护每个时间点系统状态上的概率分布。从一个时间点到下一个时间点，Figaro 考虑模型的动态性和新观测值的条件，产生新状态上的概率分布

这一过程有不同的名称，包括监控、状态估计和过滤。这些名称指的都是同一过程。实现这一过程的算法常常称为**过滤算法**，Figaro 中使用的就是这个名称。**粒子过滤**可能是最流行的过滤算法。这是一种抽样算法，用一组样本或者“粒子”表现每个时点系统状态的分布。关于粒子过滤算法的细节可以在第 12 章中找到。

为了在 Figaro 中创建粒子过滤算法，传递 3 个参数：

- 初始宇宙
- 将前一个宇宙带入下一个宇宙的函数
- 每个时间步使用的粒子数

在我们的饭店示例中，创建如下的算法：

```
val alg = ParticleFilter(initial, nextUniverse, 10000)
```

下一步和其他 Figaro 算法一样：使用 alg.start()启动算法。对于粒子过滤，这会生成初始状态的概率分布。

粒子过滤的大部分工作由 advanceTime 方法完成，该方法将系统从一个时间步推进到下一个时间步，同时考虑新的证据。在本例中，调用如下语句：

```
alg.advanceTime(evidence)
```

证据的描述方法与之前看到的不同。在本书中，您已经看到了通过为元素添加条件或者约束而指定的证据。在此不能使用这种方法，因为您没有表示任一时点系统状态的元素的直接句柄；它们是 nextUniverse 函数的内部变量。作为替代，您按照名称引用这些元素。所以，必须告诉粒子过滤器您有证据的元素名称，以及证据的特性。

例如，您可以这样指定证据：

```
NamedEvidence("arriving", Observation(3))
```

这是 NamedEvidence 类的一个实例，它将一个名称（arriving）和一部分证据（这里是名为 arriving 的元素值为 3 的观测结果）配对。这部分证据也可以是更普遍的条件或者约束。粒子过滤 advanceTime 方法的参数是这些 NamedEvidence 条目的列表，该列表指定了该时间点新接收的所有证据。

在我们的饭店示例中，假定饭店老板断断续续地观察一个时间步中到达饭店的人数。所以，她在任意特定时间步都可能提供或者不提供证据。在 Scala 中，您可以为她提供一个 Option[Int]型变量，这个变量可以为 None，或者观察到的人数。然后，将这个可选观测值转换为 advanceTime 的参数。下面是相关的代码：

```
val evidence = {
  arrivingObservation(time) match {
    case None => List()
    case Some(n) => List(NamedEvidence("arriving", Observation(n)))
  }
}
alg.advanceTime(evidence)
```

调用 advanceTime 之后，您可以查询当前时点的系统状态。粒子过滤为此提供多种方法，如 currentProbability、 currentExpectation 和 currentDistribution。同样，因为没有特定元素的句柄，您按照名称引用被查询元素。例如，为了得到等位客人数大于 4 的概

率，您可以使用如下代码：

```
alg.currentProbability("waiting", (i: Int) => i > 4)
```

要得到饭店中就坐客人的预期（平均）数量，可以调用：

```
alg.currentExpectation("seated", (l: List[Int]) => l.length)
```

下面总结了在给定初始宇宙和从前一宇宙返回下一宇宙的函数情况下运行粒子过滤所需的步骤。

1．创建粒子过滤。

2．启动粒子过滤获得初始分布。

3．对于每个时间步完成如下工作。

a）收集该时间步的证据。

b）以该证据调用 advanceTime，获得新状态上的分布。

c）查询新分布。

您可以在本书提供的代码中看到这些步骤，产生的输出如下：

```
Time 1: expected customers = 9.6498, expected waiting = 1.2325
Time 2: expected customers = 9.2651, expected waiting = 0.7388
Time 3: expected customers = 9.3622, expected waiting = 1.2295
Time 4: expected customers = 9.4169, expected waiting = 1.7192
Time 5: expected customers = 9.6011, expected waiting = 2.0629
Time 6: expected customers = 9.5866, expected waiting = 2.4307
Time 7: expected customers = 8.9794, expected waiting = 1.3958
Time 8: expected customers = 9.489, expected waiting = 2.2148
Time 9: expected customers = 9.5205, expected waiting = 2.5118
Time 10: expected customers = 9.5373, expected waiting = 2.8227
Time 11: expected customers = 9.5656, expected waiting = 3.1624
Time 12: expected customers = 9.4327, expected waiting = 2.6614
```

您可以看到这个程序是如何随时监控就坐和等位的顾客的。您的目标已经实现，您建立了一个复杂动态系统的模型，可以随时监控该系统的状态。

这也就完成了本书有关建模技术的这一部分。我们的焦点是最重要的思路和最常见的用例，现在，您应该已经拥有足够的技术，可以将 Figaro 用于大部分应用，并且理解了与建模相关的 Scaladoc 的各个方面。

注意：我没有试图全面地介绍 Figaro 的每个特性。Scaladoc 提供了 Figaro 完整功能的描述。如果您有什么不清楚的地方，请发送邮件到 figaro@cra.com；我们始终渴望着改善文档。

本书的下一部分介绍概率推理的工作原理，帮助您最大限度地利用模型。

8.4　小结

■ 动态系统由随时间推移而改变的状态组成；不同时点的状态相互关联。

- 许多动态概率模型采用马尔科夫假设——在前一个状态给定的情况下，当前状态条件独立于所有更早的状态。
- 尽管隐含马尔科夫模型采用马尔科夫假设，但是在推理当前状态时必须考虑所有之前的证据，因为前一个状态无法观察到。
- 动态贝叶斯网络就像有多个状态变量的隐含马尔科夫模型扩展；通过将这些变量的类型变为丰富的数据结构，可以建立结构随时变化的系统模型。
- 监控或者过滤是根据接收到的观测值随时跟踪系统状态的过程；这一过程使用粒子过滤等算法实现。
- Figaro 中的过滤通过创建一个初始宇宙和将前一个宇宙映射到下一个宇宙的函数实现。
- 在 Figaro 中进行过滤时，影响下一时间步的元素以及证据和查询元素按名引用。

8.5 练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．在乒乓球比赛中，先得 21 分者为胜。Alice 和 Bob 进行一场比赛，Alice 实力稍强，所以得分概率为 52%。使用马尔科夫链建立乒乓球比赛的模型，并用它估算 Bob 赢得比赛的概率。

2．现在，我们来尝试练习 1 的一个变种。Alice 和 Bob 的相对技能水平未知，比赛开始时我们假定 Alice 得分的概率是一个 Beta(2，2)分布。这也可以马尔科夫链的形式建模，只是迁移概率取决于一个未知的参数。根据当前比分是 Alice 以 11∶8 领先这一事实，使用该模型计算 Bob 取胜的概率。

3．让我们建立一名学生通过 10 个难度不断加大的章节学习的模型。每一章都有一个测验。学生的测验成绩取决于她从这一章中学到了多少。此外，每个章节互为基础，所以学生从某一章中学到多少取决于从前一章中学到多少。使用隐含马尔科夫模型建立这种情况的模型，根据学生通过前三次测验的事实，预测她成功通过最后一次测验的概率。

4．使用 DBN 创建一个公司的简单经济模型。这个 DBN 有 3 个变量：投资、利润和资本。在任何时间步，资本等于之前的资本加新利润，然后减去新投资。任何时间步的利润总额不确定，但是倾向于随投资的增加而增加。考虑给定时间点的投资等于前一时间点总资本额固定比例的不同策略。对于某个给定的固定起始资本总额，预测 10 个时间步之后不同投资策略得到的资本总额。

5．让我们来创建一个网络随时演变的简单模型。在每个时间点，发生如下两种情况之一：网络中添加一个新节点并随机地连接到现有的一个节点，这种情况的概率为

0.1；两个现有节点的边状态反转（如果之前两者之间没有边，则添加一条边；否则删除边），这种情况的概率为 0.9。

使用某个给定的初始网络，创建一个 Figaro 模型，表示 100 个时间步中的网络演化。预测 100 步之后网络中的边数。

6．让我们来创建一个新歌在流行排行榜上能否取得成功的模型。在歌曲发行时，越来越多的人会接触到它，所以流行程度不断提高。在某个时点，对歌曲感兴趣的大部分人已经接触到它，所以购买者减少，在流行排行榜上的排名也随之下降。

我们将创建一个有 5 个变量的模型。Quality 代表歌曲的总体质量；Newly Exposed 是在给定的一周中第一次听到这首歌的人数；Total Exposed 是听过这首歌的总人数；Newly Bought 是在给定的一周中购买歌曲的人数；Total Bought 是目前为止购买过这首歌曲的人数。Quality 不会随时间的推移而变化，而 Total Bought 是前一个 Total Bought 加上 Newly Bought，Total Exposed 也类似。Newly Bought 依赖于 Quality 和 Newly Exposed；新接触的人越多，购买歌曲的人就越多，但是这还取决于歌曲的质量。最后，Newly Exposed 依赖于 Total Exposed 和 Newly Bought；听过歌曲的人越多，新的聆听者就越少；另一方面，在给定的一周内购买歌曲的人越多，在广播中出现的次数越多，听过歌曲的人也就越多。

因为歌曲停留在排行榜上的时间没有限制，我们必须使用 Figaro 宇宙创建持续模型。Newly Bought 是观测变量。我们将查询 Total Exposed。编写一个 Figaro 程序表达该模型。

a）从这个模型中生成数据。使用粒子过滤在没有证据的情况下创建 Newly Bought 数值序列。当 Newly Bought 低于某个阈值（对应于歌曲从排行榜上滑落）时停止。

b）此时，使用生成的数据，观测 Newly Bought 并估算一段时间的 Total Exposed。

第 3 部分

推理

您已经编写了一个概率程序，如何使用它呢？您必须应用一个推理算法。为了最大限度地利用概率编程，您必须理解推理算法以及使用它们的最好方式。本书的第 3 部分专门介绍推理。第 9 章提供概率推理的基本概念。随后的第 10～13 章描述各种推理算法。这些章节在算法的理论描述和使用算法中的实际考虑之间达成平衡，并提供了在现实世界中的示例。

您将学习推理算法的两个主要家族：因子分解算法和抽样算法。掌握这两类算法的原理有助于理解概率编程中遇到的大部分算法。相应地，第 10 章的重点是因子分解算法，第 11 章则聚焦于抽样算法。第 12 章说明如何采用第 10 和 11 章中学习到的算法回答各种查询。最后，第 13 章介绍了两种高级且非常重要的推理：关于动态系统的推理，以及模型数值参数的学习。

第 9 章　概率推理三原则

本章介绍如下内容：

■　使用概率模型的 3 个重要原则

——　链式法则帮助您从简单的组件构建复杂模型

——　全概率公式帮助您简化复杂概率模型以回答简单的查询

——　贝叶斯法则可以用于从结果的观测得出关于根源的结论

■　贝叶斯建模基础，包括如何从数据估算模型参数，并用它们预测未来的情况

在本书的第 2 部分中，您学到了编写应用于各种情况的概率程序的相关知识。您知道概率编程系统使用运行于这些程序之上的推理算法，根据证据回答查询。它们是如何做到的？这就是这一部分的内容。重要的是，了解了这些原理，就能够设计模型，选择支持快速、精确推理的算法。

本章从推理的基础知识开始：概率推理的三原则。三原则的输入和输出如图 9-1 所示。

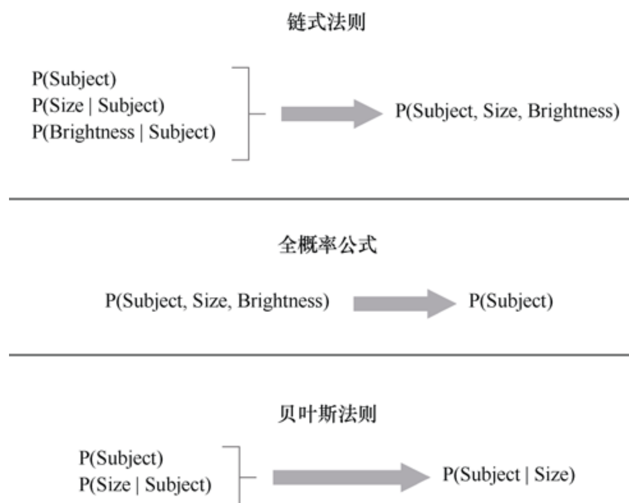


图 9-1 概率推理三原则的输入和输出。链式法则帮助您将一组条件概率分布转换为联合概率分布。全概率公式帮助您从一组变量的联合概率分布生成单一变量上的分布。贝叶斯法则帮助您在原因给定的情况下，将结果的条件概率分布“反转”为在结果给定情况下原因的条件概率分布

- 您首先将学习链式法则，该法则帮助您从简单（单独变量的局部条件概率分布）到复杂（所有变量的完整联合概率分布）。
- 9.2 小节中描述的全概率公式，从复杂（完整联合分布）回到简单（单一变量的分布）。
- 最后是 9.3 小节介绍的贝叶斯法则，这可能是最著名的推理原则。贝叶斯法则可以“反转”依赖性的方向，将给定原因的结果条件分布转化为给定结果的原因分布。贝叶斯法则对于加入证据（通常是结果的观测值）推导出原因至关重要。

这三条推理原则可以用于回答查询。

深入新材料的学习之前，我们先复习一下第 4 章中的一些定义，在本章的学习中需要它们。

- **可能世界**——您认为可能的所有状态。
- **概率分布**——为每个可能世界指定的 0～1 的概率，所有概率加起来为 1。
- **先验概率分布**——在看到任何证据之前的概率分布。

- **证据调节**——将证据应用到某个概率分布的过程。
- **后验概率分布**——看到证据之后的概率分布，调节的结果。
- **条件概率分布**——为其他变量值的每个组合指定某个变量上概率分布的规则。
- **规格化**——按比例调整一组数量，使之加总为 1 的过程。

注意：对于每条原则，我们都用一段补充材料介绍了通用的数学定义。如果您希望深入理解它们，这些材料是很有用的；如果您熟悉数学标记法，这种更为抽象的讨论有助于帮助您巩固对原理的理解。如果不是这样，您尽可以跳过这些补充材料，我们的主要目的是帮助您理解使用这些规则的原因和方法。

9.1　链式法则：从条件概率分布构建联合分布

您可能记得，第 4 章介绍概率模型如何定义可能世界上的概率分布，以及概率模型的成分：变量、依赖性、函数形式和数值参数。我曾经提示过，链式法则是将这些成分转化为可能世界上概率分布的必要机制。我承诺将在第 3 部分对链式法则进行全面的讨论，现在到时候了。

链式法则如何定义可能世界上的概率分布？换言之，**如何为每个可能世界指定一个 0 和 1 之间的数值**？让我们回到第 4 章中的伦勃朗示例。从变量 Subject、Size 和 Brightness 开始，假定您有一个依赖模型，Size 和 Brightness 均依赖于 Subject，但是相互之间没有依赖。您还得到了 Subject 概率分布的规格；在 Subject 给定情况下 Size 的 CPD；以及 Subject 给定情况下 Brightness 的 CPD。这些成分在图 9-2 中总结。（Subject 没有依赖任何其他变量，我使用和其他变量相同的 CPD 表格，只是表格中没有条件变量，只有一行。）

现在，可能世界为 Subject、Size 和 Brightness 变量各指定一个值。**如何得到可能世界的概率**？例如，P(Subject = People, Size = Large, Brightness = Dark)等于多少？根据链式法则，这很容易计算。**找出 CPD 表格中的对应条目，将它们相乘**。在本例中：

P(Subject = People, Size = Large, Brightness = Dark) =

P(Subject = People) × P(Size = Large | Subject = People) × P(Brightness = Dark | Subject = People) =

0.8 × 0.5 × 0.8 =

0.32

您可以对 Subject、Size 和 Brightness 的所有可能值使用相同的公式，获得表 9-1 中所示的结果。这个结果称为 Subject、Size 和 Brightness 的联合概率分布，因为它指定了这 3 个变量值每个组合的概率。

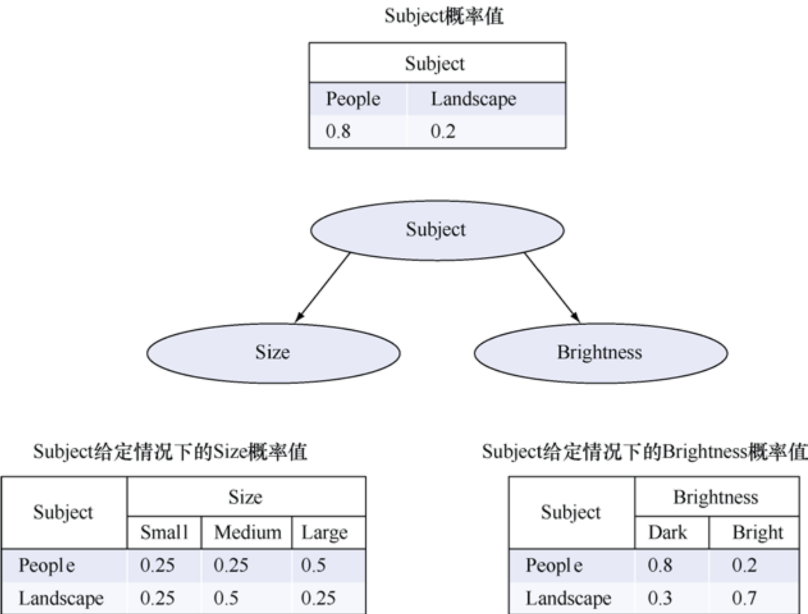


图 9-2　链式法则示例的贝叶斯网络结构和 CPD

表 9-1　对图 9-2 中的 CPD 应用链式法则得到的联合概率分布。您将 P(Subject)乘以 P(Size | Subject)和 P(Brightness | Subject)。这些概率的总和为 1

Subject	Size	Brightness	概　　率
People	Small	Dark	0.8 × 0.25 × 0.8 = 0.16
People	Small	Bright	0.8 × 0.25 × 0.2 = 0.04
People	Medium	Dark	0.8 × 0.25 × 0.8 = 0.16
People	Medium	Bright	0.8 × 0.25 × 0.2 = 0.04
People	Large	Dark	0.8 × 0.5 × 0.8 = 0.32
People	Large	Bright	0.8 × 0.5 × 0.2 = 0.08
Landscape	Small	Dark	0.2 × 0.25 × 0.3 = 0.015
Landscape	Small	Bright	0.2 × 0.25 × 0.7 = 0.035
Landscape	Medium	Dark	0.2 × 0.5 × 0.3 = 0.03
Landscape	Medium	Bright	0.2 × 0.5 × 0.7 = 0.07
Landscape	Large	Dark	0.2 × 0.25 × 0.3= 0.015
Landscape	Large	Bright	0.2 × 0.25 × 0.7 = 0.035

真相是，我要了点小花招。标准的 3 变量链式法则规定，对于第三个变量，您必须同时根据前两个变量调节其概率。所以，我们不使用如下的计算公式：

P(Subject = People, Size = Large, Brightness = Dark) =

P(Subject = People) × P(Size = Large | Subject = People) × P(Brightness = Dark | Subject = People)

而应该这么计算：

P(Subject = People, Size = Large, Brightness = Dark) =

P(Subject = People) × P(Size = Large | Subject = People) × P(Brightness = Dark |

Subject = People, Size = Large)

这才是链式法则的正式陈述。但是我利用了有关依赖性的特殊知识——Brightness不依赖于 Size，只依赖于 Subject。在 Subject 给定的情况下，Brightness 条件独立于 Size：

P(Brightness = Dark | Subject = People, Size = Large) =

P(Brightness = Dark | Subject = People)

因此，按照我的方法简化链式法则是合理的。只要您有一个贝叶斯网络，希望使用链式法则定义所有变量上的完整概率分布，总是可以简化这条法则，使得每个变量只依赖于网络中的父变量。反之，如果 Brightness 在 Subject 给定情况下不条件独立于 Size，您就不得不使用较长的形式。贝叶斯网络和链式法则是相辅相成的。贝叶斯网络规定了用于计算联合分布的链式法则形式。

链式法则的内容就是这些，它在概率建模中是简单而关键的原则。链式法则不仅对理解贝叶斯网络是必不可少的，一般来说也是其中的生成模型。因为概率程序是生成模型的编码，既然理解了链式法则，就有了理解概率程序所定义的概率模型的基础。

链式法则的一般形式

链式法则是适用于任何依赖模型和任何变量 CPD、任何函数形式的通用原则。只要每个变量都有规定在所依赖变量任何可能值下概率分布的 CPD，就可以将 CPD 中对应数值相乘得出所有变量的联合概率。

在数学标记法中，您从两个变量 X 和 Y 开始，Y 依赖于 X。您得到 X 值上的概率分布 $P(X)$ 和给定 X 情况下 Y 的 CPD——$P(Y \mid X)$。链式法则取得这两个成分，将其转化为 X 和 Y 的联合概率分布 $P(X, Y)$。对于 X 的每个可能值 x 和 Y 的每个可能值 y，链式法则可以用如下简单公式定义：

$$P(X = x, Y = y) = P(X = x)P(Y = y \mid X = x)$$

标记法警告：使用 X 和 Y 这样的大写字母表示变量，x 和 y 等小写字母表示变量值，是一种标准的做法。

您可以用简单的方式表示该公式对每个 x 和 y 均成立：

$$P(X, y) = P(X)P(Y \mid X)$$

这个容易记忆的公式是用于特定 x 和 y 值的许多公式的简写。

如果变量超过两个怎么办？链式法则可以推广到任意数量的变量。假定您有变量 X_1、X_2、… X_n。在链式法则的标准陈述中，您不做出任何独立性的假设，所以每个变量都依赖于之前的所有变量。用简写标记法表达的完整链式法则如下：

$$P(X_1, X_2, \ldots X_n) = P(X_1)P(X_2 \mid X_1)P(X_3 \mid X_1, X_2)\ldots P(X_n \mid X_1, X_2, \ldots X_{n-1})$$

我们来看看这个公式所表达的含义。它说明，为了得到 X_1、X_2、…X_n 上的联合概率分布，从 X_1 开始，得到它的概率，然后观察依赖于 X_1 的 X_2，从 CPD 中得到它的对应概率。接着从 CPD 获得依赖于 X_1 和 X_2 的 X_3 概率，循环继续直到最后从 CPD 得到取决于所有之前变量的 X_n 概率。顺便说一下，这个公式正是链式法则名称的由来。您从一个条件分布链计算出联合概率分布。

我们的多变量链式法则公式对依赖性不做任何假设，特别是没有独立关系。增加独立性信息可以显著简化该公式。您不需要在给定变量的“|”右侧包含所有之前的变量，只需要包含在依赖性模型中直接依赖的变量。例如，考虑 3 个变量 Subject、 Size 和 Brightness 的情况，如果遵循上面的公式，得到的结果如下：

P(Subject, Size, Brightness) = P(Subject) P(Size | Subject) P(Brightness | Subject, Size)

但是根据网络，Brightness 不依赖于 Size，只依赖于 Subject。所以，公式可以简化为：

P(Subject, Size, Brightness) = P(Subject) P(Size | Subject) P(Brightness | Subject)

确实，这就是用于计算表 9-1 的公式。

9.2 全概率公式：从联合分布获得简单查询结果

链式法则帮助您从简单的 CPD 中构建一个联合分布，例如 Subject、Size 和 Brightness 的联合分布。通常，您的查询是关于一个特定变量或者少数几个变量的。例如，您可能想要根据对一幅画作的观察，推断画家的身份。假定您有所有变量上的一个联合分布。如何得到单一变量上的概率分布？原理很简单：任何变量值的概率等于符合该值的所有变量的联合概率的总和。

您已经在第 4 章中看到这一基本原理：任何事实的概率是与该事实一致的可能世界概率的总和。所以，要获得 Subject = Landscape 的概率，只需要加总所有与 Subject = Landscape 一致的可能世界的概率。因为每个可能世界由所有变量（包括 Subject）的一个组合构成，您只需要搜索 Subject 赋值为 Landscape 的可能世界。这个简单原理通常被称作**全概率法则，**但是我更愿意使用**全概率公式**这一普通的名称。

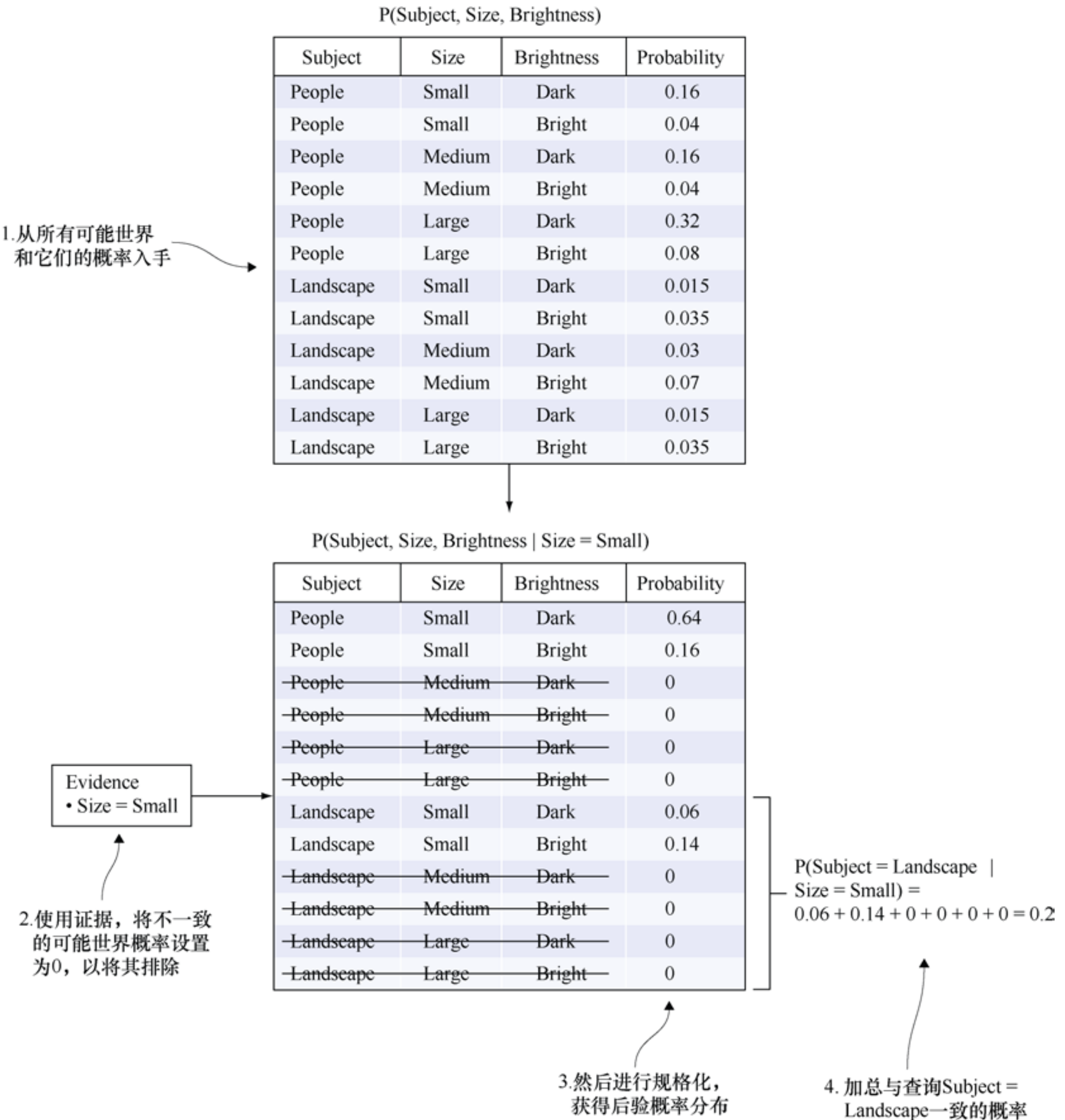
全概率公式的使用如图 9-3 所示。首先得到图中顶部的先验概率分布。然后根据 Size=Small 证据进行调节，获得中间的后验概率分布。您使用两个平常的步骤：首先，删除与 Size=Small 这一证据不相符的变量值，然后规格化其余概率使其总和为 1。在该图的底部，您使用全概率公式计算给定证据下画作是风景画的概率。您将 Subject 变量为 Landscape 的所有行对应的概率相加。

注意，Size 取 Small 之外的值的那一行的后验概率为 0。这是因为您删除了与证据不符的可能世界并将其概率设置为 0。所以，实际上

P(Subject = Landscape, Brightness = Dark | Size = Small)

等价于

P(Subject = Landscape, Brightness = Dark, Size = Small | Size = Small)



P(Subject, Size, Brightness)

Subject	Size	Brightness	Probability
People	Small	Dark	0.16
People	Small	Bright	0.04
People	Medium	Dark	0.16
People	Medium	Bright	0.04
People	Large	Dark	0.32
People	Large	Bright	0.08
Landscape	Small	Dark	0.015
Landscape	Small	Bright	0.035
Landscape	Medium	Dark	0.03
Landscape	Medium	Bright	0.07
Landscape	Large	Dark	0.015
Landscape	Large	Bright	0.035

P(Subject, Size, Brightness | Size = Small)

Subject	Size	Brightness	Probability
People	Small	Dark	0.64
People	Small	Bright	0.16
~~People~~	~~Medium~~	~~Dark~~	0
~~People~~	~~Medium~~	~~Bright~~	0
~~People~~	~~Large~~	~~Dark~~	0
~~People~~	~~Large~~	~~Bright~~	0
Landscape	Small	Dark	0.06
Landscape	Small	Bright	0.14
~~Landscape~~	~~Medium~~	~~Dark~~	0
~~Landscape~~	~~Medium~~	~~Bright~~	0
~~Landscape~~	~~Large~~	~~Dark~~	0
~~Landscape~~	~~Large~~	~~Bright~~	0

图 9-3　使用全概率公式回答查询。某个变量值的总概率是所有与该值一致的联合概率的总和

您可以看到 P(Subject = Landscape | Size = Small)是图 9-3 中间表格内两行的总和，可以用如下运算表示：

P(Subject = Landscape | Size = Small) =

P(Subject = Landscape, Brightness = Dark | Size = Small) + P(Subject = Landscape, Brightness = Bright | Size = Small)

上述累加运算的简洁写法使用希腊字母Σ，该字母是标准的数学累加标记：

P(Subject = Landscape | Size = Small) =

Σ_b P(Subject = Landscape, Brightness = b | Size = Small)　　（1）

在公式的右侧，b 代表 Brightness 的任何可能值，Σ_b 表示累加 Brightness 所有可能值下的后续项值。假定我们要计算 Brightness 的总和。现在，公式（1）适用于 Subject 和 Size 的任意可能值，所以可以使用前一小节的简写方式

P(Subject | Size) = Σ_b P(Subject, Brightness = b | Size)

全概率公式的一般形式

您已经看到了全概率公式在我们的例子中的应用，为了解一般数学定义做好了准备。同样是这个简单的原理，但是标记法稍微凌乱一些。您有一组变量上的联合概率分布，希望加总某些变量，获得在其他变量上的分布。例如，您可能有 Color、 Brightness、 Width 和 Height 上的联合分布，希望得到 Color 和 Brightness 上的分布，加总 Width 和 Height。现在，您在所有变量上的联合分布可以根据其他变量集（如 Rembrandt 和 Subject）调节。为了保持公式的简短，您将使用每个变量的首字母。还有，我们假设 Width 和 Height 的可能值为 small（小）和 large（大）。根据全概率公式：

P(C = yellow, B = bright | R = true, S = landscape) =

P(C = yellow, B = bright, W = small, H = small | R = true, S = landscape) +

P(C = yellow, B = bright, W = small, H = large | R = true, S = landscape) +

P(C = yellow, B = bright, W = large, H = small | R = true, S = landscape) +

P(C = yellow, B = bright, W = large, H = large | R = true, S = landscape)

您可以用数学标记法写出上述公式。我们将您希望求取分布的变量称作 $X_1,\cdots X_n$，需要加总的变量称作 $Y_1,\cdots Y_m$，调节所依据的变量称作 $Z_1, \cdots Z_l$。全概率公式规定，对于 $X_1,\cdots X_n$ 的任何值 $x_1, \cdots x_n$，$Z_1, \cdots Z_l$ 的任何值 $z_1, \cdots z_l$：

$$P(X_1 = x_1,\ldots,X_n = x_n \mid Z_1 = z_1,\ldots,Z_l = z_l) =$$

$$\Sigma_{y_1} \Sigma_{y_2} \ldots \Sigma_{y_m} P(X_1 = x_1,\ldots,X_n = x_n, Y_1 = y_1,\ldots,Y_m = y_m \mid Z_1 = z_1,\ldots,Z_l = z_l)$$

这些公式说明，要取得目标变量 $X_1,\cdots X_n$ 取值为 $x_1, \cdots x_n$ 的概率，必须加总所有变量上的全条件分布中目标变量取值与 $x_1, \cdots x_n$ 相符的所有情况。

因为这个公式适用于所有值 $x_1, \cdots x_n$ 和 $z_1, \cdots z_l$，可以使用和 9.1 小节中相同的简写：

$$P(X_1,\ldots,X_n \mid Z_1,\ldots,Z_l) = \Sigma_{y_1} \Sigma_{y_2} \ldots \Sigma_{y_m} P(X_1,\ldots,X_n, Y_1 = y_1,\ldots,Y_m = y_m \mid Z_1,\ldots,Z_l)$$

另一种标记技巧可以使全概率公式更容易记忆。如果有一组变量 $X_1,\cdots X_n$，您可以用粗体的 $\boldsymbol{X}$ 表示它们。这样 $\boldsymbol{X}$ 就是 $X_1,\cdots X_n$ 的简写形式。同样，可以使用粗体的 $\boldsymbol{x}$ 作为值 $x_1, \cdots x_n$ 的简写形式。

标记法警告：使用未加粗的斜体字母如 X 和 x 表示单独变量或者值，粗体的 $\boldsymbol{X}$ 和 $\boldsymbol{x}$ 表示一组变量或者值的做法很常见。

所以，对于特定值 x 和 z：

$$P(X = x \mid Z = z) = \Sigma_y P(X = x, Y = y \mid Z = z)$$

推广到所有值 x 和 z，最终得到如下精练的公式：

$$P(X \mid Z) = \Sigma_y P(X, Y = y \mid Z)$$

上述公式总结了全概率公式。

从一组变量的联合分布入手，累加一些变量以获得其余变量上的概率分布时，可能会遇到一个术语。这样得出的分布被称作其余变量上的**边缘分布**，加总变量以获得其他变量上边缘分布的过程被称作**边缘化**。最典型的情况是加总除了某个变量之外的所有变量，得到该变量上的边缘分布。

现在，您已经了解了概率推理三原则中的两个。下面我们将注意力转向最后一个原则，这可能是最有趣的一个原则。

9.3 贝叶斯法则：从结果推断原因

概率模型推理拼图的最后一块是贝叶斯法则，这一法则是以 18 世纪数学家托马斯•贝叶斯的姓氏命名的，后者第一个发现了如何由对结果的观测推导出有关原因的知识。**贝叶斯法则帮助您结合原因的先验概率（在知道任何关于结果的情况之前）和给定原因下该结果的概率，计算在给定结果的情况下，某个原因的条件概率。**

9.3.1 理解、原因、结果和推理

贝叶斯法则与因果的概念相关，后者又与模型中的依赖性相关。在常规程序中，当变量 X 的定义中使用了另一个变量 Y，改变 Y 的值可能造成 X 值的变化。所以，在某种意义上，Y 是 X 的根源（原因）。同样，如果构造一个 X 依赖于 Y 的概率模型，Y 往往是 X 的根源。例如，考虑 Subject 和 Brightness。在您的模型中 Brightness 依赖于 Subject，通常画家可能在决定鲜艳度之前决定画作的类型。所以在这个意义上，Subject 是 Brightness 的根源。

在这里使用**根源（原因）**这个词有些不太精确。更准确的描述是：您建立数据**生成过程**的模型。在这个过程中，您想象画家首先选择主题，然后根据主题选择鲜艳度。所以画家首先生成 Subject 变量的值，然后传递给 Brightness 变量值的生成过程。当模型遵循某种生成过程时，您将在一个变量的值为另一个变量所用时宽泛地使用**原因**和**结果**这两个词。

图 9-4 用贝叶斯网络描述生成过程的一个较为复杂的例子。在这个例子中，生成的第一个变量是“画作是否伦勃朗的作品”，因为画家的身份影响关于画作的任何特征。然后，画家选择 Subject，后者进而帮助确定 Size 和 Brightness。Size 依赖于 Rembrandt 和 Subject 的原因是不同画家的风景画倾向于采用不同的尺寸；Brightness 的情况也类似。

图 9-4 的右边指出重要的一点。尽管生成过程遵循模型中的箭头，关于模型的推理可以采用任何方向。实际上，在这个例子中，我们的目标是确定画作是不是伦勃朗的，所以推理的方向与生成过程相反。我在本书中一直强调这一点；网络中箭头的方向不一定是进行推理的方向。不要让您关于某个领域的典型推理方法（例如“我观察画作的鲜艳度以确定画家是谁”）指导构造网络的方法。作为替代，应该考虑生成过程。在大部分情况下，遵循生成过程可以生成最简单、最清晰的模型。您可以根据自己的需要选择推理的方向。

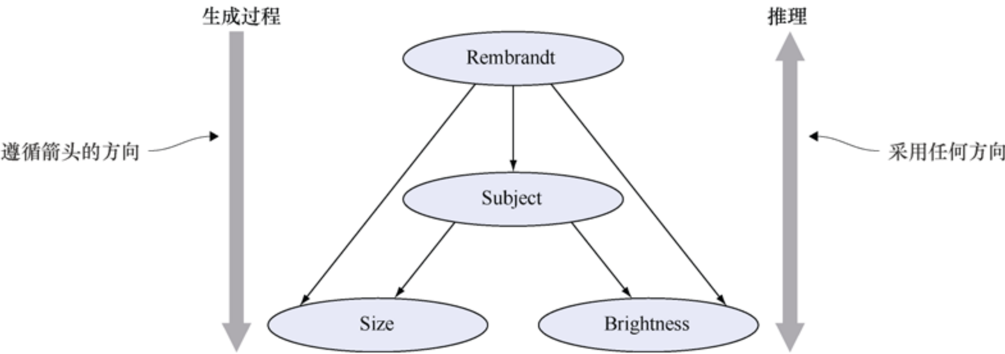
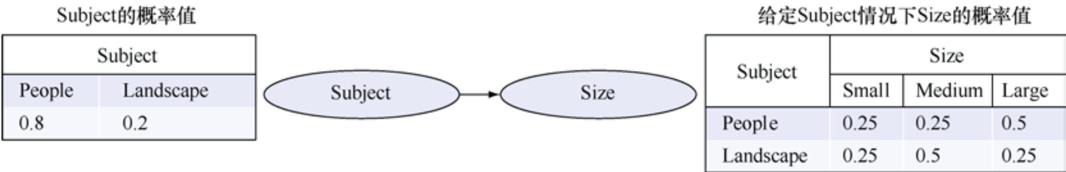


图 9-4 网络箭头往往遵循生成过程的方向，但是推理可以采用任何方向

我已经说过，您可以从网络中箭头的相反方向进行推理。怎么做呢？贝叶斯法则就是答案！让我们来看看图 9-5 中的双变量示例。这里，网络遵循自然的生成过程，Subject 决定 Size。您得到了 P(Subject) 和 P(Size|Subject)两个成分。首先，我们考虑向前推理——按照生成过程的方向。假定您观察到 Subject = Landscape，希望查询 Size 的后验概率，可以从 P(Size | Subject)直接得到。如果想要从关于原因的证据推理出结果，立刻就可以得到该信息。



Subject	
People	Landscape
0.8	0.2

Subject	Size		
	Small	Medium	Large
People	0.25	0.25	0.5
Landscape	0.25	0.5	0.25

图 9-5 贝叶斯法则示例的双变量模型

但是，您往往观察到的是关于结果的证据，希望推导该结果可能原因的相关信息。您希望反转模型，因为想要得到的是 P(Subject | Size)，即结果给定情况下某种原因的概率。贝叶斯法则使之成为可能。

9.3.2　实践中的贝叶斯法则

贝叶斯法则的运算很简单。我将首先展示其工作原理，然后解释每个步骤。完整的过程如图 9-6 所示。您从图 9-5 中的模型入手，然后观察到证据 Size = Large。您希望计算这个证据下 Subject 的后验概率分布——计算 P(Subject | Size = Large)。下面是计算过程。

1．计算 P(Subject = People) P(Size = Large | Subject = People) = 0.8 × 0.5 = 0.4，P(Subject = Landscape) P(Size = Large | Subject = Landscape) = 0.2 × 0.25 = 0.05。这些数字显示在图 9-6 中间的表格里。

2．规格化这个表格得到所需的答案。规格化因子是 0.4 + 0.05 = 0.45。所以 P(Subject = People | Size = Large) = 0.4 / 0.45 = 0.8889，P(Subject = Landscape | Size = Large) = 0.05 / 0.45 = 0.1111。这个答案显示在图 9-6 中下方的表格里。

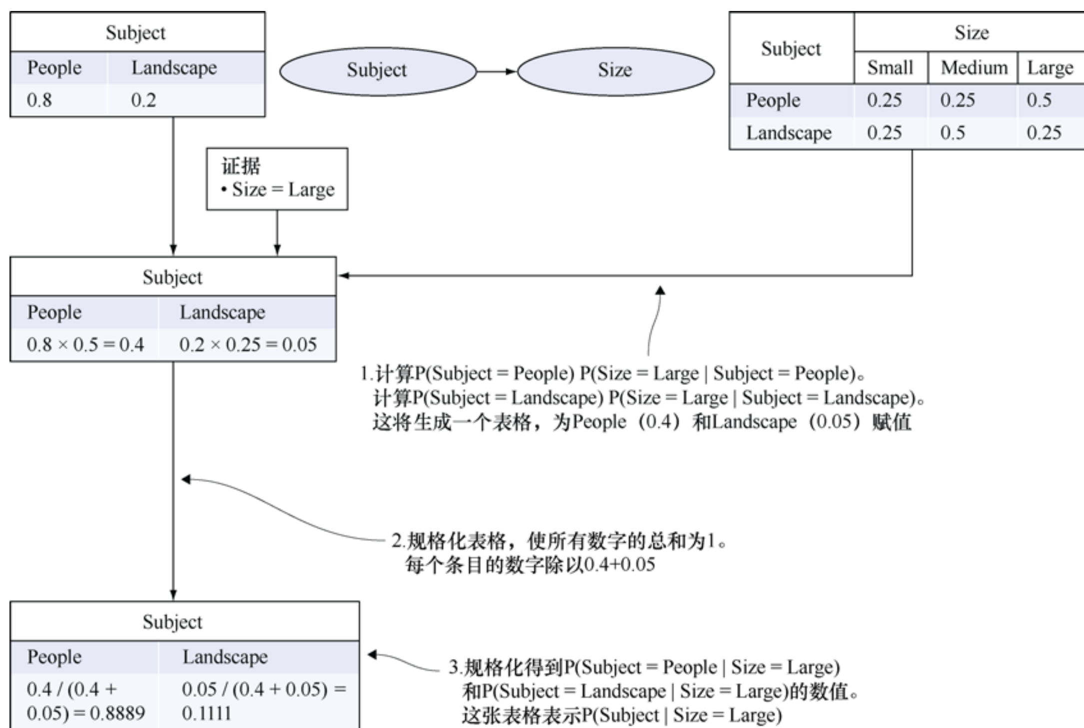


图 9-6　贝叶斯法则的运算

那么，为什么上述计算是合理的呢？在这一过程中，您拥有了链式法则和全概率公式的成分，将要使用 9.1 小节中学到的链式法则，从 CPD 成分中构造出联合概率分布。然后，您将再次应用链式法则，不过这次的方向相反。最后，您将使用全概率公式完成计算。下面是计算步骤。

1．取得 P(Subject)和 P(Size | Subject)，应用链式法则得到 P(Subject, Size) = P(Subject) P(Size | Subject)。

2．使用链式法则，但是从反方向进行，可以写作 P(Size, Subject) = P(Size) P(Subject | Size)。

3．因为 P(Subject, Size) 和 P(Size, Subject)相等，可以将 1 和 2 结合起来，得到 P(Size) P(Subject | Size) = P(Subject) P(Size | Subject)。

4．将等式两端同时除以 P(Size)，得到

$$P(\text{Subject}|\text{Size})=\frac{P(\text{Subject})P(\text{Size}|\text{Subject})}{P(\text{Size})}$$

现在，公式的左侧就是查询的答案 P(Subject | Size)。这个公式通常被称作贝叶斯法则，但是它的形式还不实用，因为它包含了您尚未得到的 P(Size)，所以还需要一个步骤。

1．运用全概率公式和链式法则，用已知项表示 P(Size)。首先，运用全概率公式得出 P(Size) = Σ_s P(Subject = s, Size)，然后，使用链式法则得出 P(Subject = s, Size) = P(Subject = s) P(Size | Subject = s)。最后，组合上式得出 P(Size) = Σ_s P(Subject = s) P(Size | Subject = s)。

2．得到最后的答案：

$$P(\text{Subject}|\text{Size=Large})=\frac{P(\text{Subject})P(\text{Size=Large}|\text{Subject})}{\Sigma_s P(\text{Subject=}s)P(\text{Size=Large}|\text{Subject=}s)}$$

您可以看到这个答案与图 9-6 中所示的两个步骤之间的关联。第一个步骤计算 Subject 两个可能值的分子 P(Subject) P(Size = Large | Subject)。现在，观察分母。您添加了用于 Subject 每个可能值 s 的 P(Subject = s) P(Size | Subject = s)。但是这只是第一步中为每个 Subject 值计算的数量。分母将第一步计算的所有数字加总。所以，您需要将每个数字除以总数。这是表达第 2 步中规格化运算的另一种方式。

虽然贝叶斯法则很简单，但是还有更多需要学习的地方。贝叶斯法则为贝叶斯建模框架（下一小节的主题）提供了基础，下一节还将更深入地探讨贝叶斯法则的工作原理，并提供贝叶斯法则一般形式的补充材料。

9.4 贝叶斯建模

贝叶斯法则为一种通用建模方法提供了基础，在这种方法中，您从结果的观察中推导出关于原因的知识，然后将这些知识应用到其他潜在结果中。

本节使用第 2 章中首次遇到的掷币场景阐述贝叶斯建模。根据 100 次掷币的结果(模型的结果)，进行如下工作。

- 使用贝叶斯法则推断硬币的偏差（结果的原因）。
- 阐述多种预测第 101 次掷币结果的方法。
  —— 最大后验（MAP）方法。
  —— 最大似然估计（MLE）方法。
  —— 全贝叶斯方法。

图 9-7 重现根据前 100 次掷币结果预测第 101 次掷币结果的贝叶斯网络。您有 3 个变量：硬币的偏差（Bias），前 100 次出现正面的数量（NumberOfHeads），以及第 101 次掷币的结果（$Toss_{101}$）。首先生成 Bias，此后所有掷币结果依赖于 Bias。如果 Bias 已知，掷币结果就是相互独立的。记住，当我说 Bias 首先生成时，描述的是生成过程，而不是说 Bias 先为人所知。这是说明变量生成的顺序不一定是推理顺序的另一个例子。在我们的例子中，Bias 先生成，但是推理的方向是从 NumberOfHeads 到 Bias。

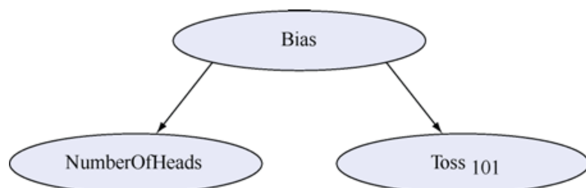


图 9-7 掷币示例的贝叶斯网络

您将使用β二项式模型，所以 Bias 的特性由一个 Beta 分布描述，而 NumberOfHeads 由一个依赖于 Bias 的二项分布描述。在此提醒：

- 二项变量描述随机过程在总尝试次数下得出某一结果的次数。在我们的例子中，用一个二项分布表示掷币结果为正面向上的次数。二项变量的参数由每次尝试得出相应结果的概率提供。
- 这个概率就是硬币的偏差。如果硬币的偏差已知，它可能是一个特定值。但是在本场景中，您不知道偏差，试图根据掷币的结果估算它。因此，您使用一个随机变量建立偏差的模型。确切地说，您使用β分布建立偏差的模型，这是一个连续分布。对于连续分布，您使用**概率密度函数**（PDF）而不是指定每个值

的概率。β分布有两个参数：α和β。α可以直观地理解为之前已经看到的正面次数加上 1。与此类似，β表示之前已经看到的背面次数加 1。正如第 4 章中所述，使用β分布是因为它和二项分布配合得很好。在本节中您将看到原因。

未来任何一次掷币结果由一个 Flip 给出，其中出现正面的概率等于 Bias。正如贝叶斯网络所暗示的那样，未来的掷币仅直接依赖于偏差。如果偏差已知，其他掷币结果不会增添任何信息。但是如果偏差未知，前 100 次掷币的结果提供了关于偏差的信息，可以用于预测第 101 次掷币的结果。

9.4.1 估算硬币的偏差

如何根据前 100 次的结果，使用这个模型预测未来掷币结果？这是贝叶斯建模的用武之地。在贝叶斯建模中，您可以使用贝叶斯法则，从观察到的正面结果次数推导出偏差的后验概率分布。然后，使用这个后验分布预测下一次掷币的结果。

这个过程如图 9-8 所示。如果观察数千次掷币，其中有 40%的结果是正面，可以推断偏差可能接近于 0.4。如果没有这么多次掷币结果，推理的置信度可能就没有那么高。这些推理是应用贝叶斯法则的直接结果。回到我们的例子，如果在 100 次掷币中有 63 次结果是正面，可以计算偏差在 NumberOfHeads = 63 情况下的分布，然后使用这个预测 $Toss_{101}$。

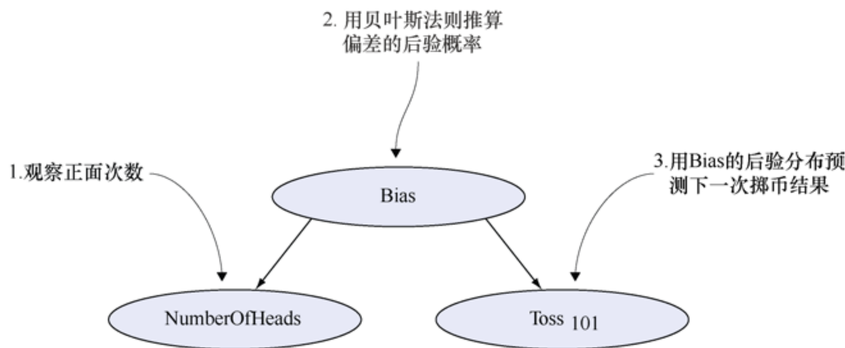


图 9-8 偏币示例中的推理顺序

为了实现这个目的，首先从 Bias 的先验分布入手。β分布的特性由两个参数α和β描述，我们将 Bias 先验β分布的参数称为α_0和β_0。回忆第 2 章，α_0和β_0表示在观察任何真实掷币结果之前，想象的正面和背面次数加上 1。为了得到后验分布，您将真实的正面和背面次数加到这些虚构的数字上。例如，假定从 beta(2, 5)开始，这意味着您想象自己已经看到了 1 次正面和 4 次背面（因为α_0是想象的正面次数加 1，β_0也类似）。然后，您观察到 63 次正面和 37 次背面的结果。偏差上的后验分布由 beta(65, 42)给出。如果将

后验β分布的参数称作α_1和β_1，可以得到如下简单公式：

$\alpha_1 = \alpha_0 +$ 观察到的正面次数

$\beta_1 = \beta_0 +$观察到的背面次数

注意：在实践中，您没有必要自己进行这些计算。概率编程系统的算法将负责这些计算。您指定自己希望使用用β-二项式模型，该模型将进行所有必要的计算。但是重要的是理解系统的工作原理，这是您花费时间阅读本节的原因。

图 9-9 展示了这个 beta(65, 42)分布，它叠加在原来的 beta(2, 5)分布之上。您可以看到两个现象。首先，分布的波峰向右移动，因为实际观差到的正面比例（63/100）高于起始的想象值（1/5）。其次，波峰变得更加陡峭。因为您有了 100 次观察，对偏差的评估更加自信。

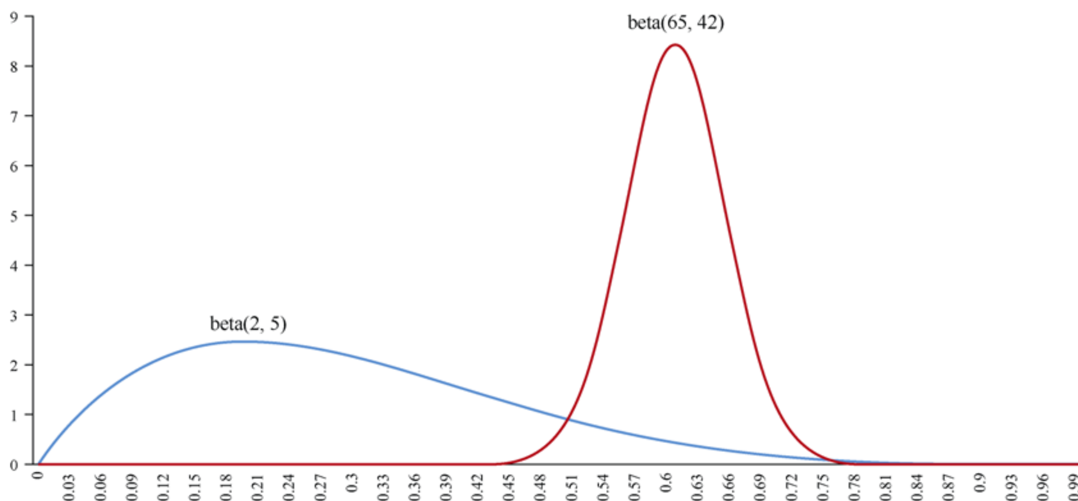


图 9-9　从一系列观测值中推算硬币的偏差。这里，您已经观察到 63 次正面和 37 次背面的结果，并将其加到α和β参数。后验 PDF beta(65,42)叠加到先验 PDF beta（2，5）上

加总结果创建用于后验分布新β分布的简单公式是应用贝叶斯法则的结果。将贝叶斯法则应用到掷币示例时，需要处理 3 个数量：

- p(Bias = *b*)：Bias 在数值 *b* 上的先验概率密度。（在这个标记法中使用小写字母 p 强调该数量是一个概率密度，而非概率。）
- P(NumberOfHeads = 63 | Bias = *b*)：给定 Bias 值 *b* 的情况下，观察到 NumberOfHeads = 63 的概率。这个概率被称作在给定数据下 *b* 的**似然率**。
- p(Bias = *b* | NumberOfHeads = 63)：Bias 值 *b* 的后验概率密度。

因为本例处理的是连续变量（偏差），比 9.3.2 小节的画作示例稍复杂一些。我将重复那个例子的结论，您可以看到这对偏币示例也适用。在 9.3.2 小节中，您得到了在给定关于画作尺寸的证据下，画作主题概率分布的如下表达：

$$P(\text{Subject}|\text{Size=Large})=\frac{P(\text{Subject})P(\text{Size=Large}|\text{Subject})}{\sum_s P(\text{Subject=}s)P(\text{Size=Large}|\text{Subject=}s)}$$

我们将重点放在分母上。它将 Subject 所有可能值下的分子值累加起来。这是一个规格化因子，确保（a）左侧始终与右侧的分子成正比；（b）左侧值的总和为 1。可以用如下标记法总结这个公式：

$$P(\text{Subject} \mid \text{Size} = \text{Large}) \propto P(\text{Subject})P(\text{Size} = \text{Large} \mid \text{Subject})$$

符号$\propto$ 的含义是左侧与右侧成**正比**，比例常数为 $1/\sum_s P(\text{Subject} = s)\ P(\text{Size} = \text{Large} \mid \text{Subject} = s)$。左侧是 Subject 上的后验概率分布。右侧的第一项是先验分布。第二项是似然率——给定 Subject 值的情况下观察到特定数据的概率。因此，前一个公式可以总结为：

后验概率 $\propto$ 先验概率 × 似然率

图 9-10 是上述公式的分解。如果关于贝叶斯建模有应该记住的公式，那就是这一个了。虽然您看到的是专门用于画作示例的公式，但是这是适用于贝叶斯法则应用的通用原则。为了得到特定值 *b* 的实际后验，对 *b* 的每个可能值计算等式右侧，并将所有结果加总得到总和 B。总和 B 是规格化因子。然后，将先验概率与似然率相乘再除以 B，得到后验概率。

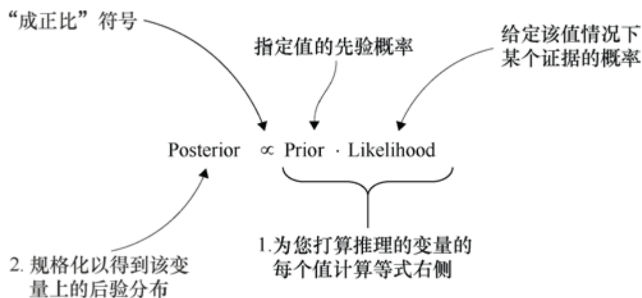


图 9-10 贝叶斯建模公式的结构

如果您考虑的是连续变量（就像我们的例子中那样），规格化过程可能很困难，因为需要求得 *b* 的所有可能值上的积分。回到掷币示例，贝叶斯法则的陈述如下：

$$P(\text{Bias=}b|\text{NumberOfHeads=63})=$$

$$\frac{P(\text{Bias=}b)P(\text{NumberOfHeads=63}|\text{Bias=}b)}{\int_0^1 P(\text{Bias=}x)P(\text{NumberOfHeads=63}|\text{Bias=}x)dx}$$

使用我们的“成正比”标记，可以重写为：

$$P(\text{Bias} = b \mid \text{NumberOfHeads} = 63) \propto P(\text{Bias} = b)P(\text{NumberOfHeads} = 63 \mid \text{Bias} = b)$$

后验概率仍然与先验概率与似然率的乘积成正比。虽然最后一个等式很简单，但是它隐藏了可能难以估算的积分。幸运的是，在β二项式模型中，这个方程式存在一个简单解，您在本节开始时已经看到了这种解法——将观察到的成功与失败次数加到β分布的参数上。这就是β与二项分布配合得很好的原因。如果采用任意的连续分布并尝试将其与二项分布配合使用，最终会遇到不容易解决的积分问题。但是当您结合β和二项分布，就很容易得到答案。

使用概率编程系统时，没有必要自行计算这些积分。概率编程系统往往能够使用近似算法处理这些困难的积分问题，所以您不会被限制在特别合适的函数形式上。不过，在有这样的函数形式时，最好使用它。

注意：在第 6 章中，您第一次遇到**共轭先验**这个术语，该术语描述和取决于参数的分布配合得很好的一个先验分布。从技术上说，这意味着后验分布和先验分布有相同的形式。运用这个术语，β分布是二项分布的共轭先验，因为参数上的后验分布也是一个β分布。当您使用共轭先验分布时，贝叶斯法则中的积分有一个简单解。这就是贝叶斯统计中频繁使用共轭分布的原因。但是在使用概率编程时，并不限于使用共轭分布。

贝叶斯法则的一般形式

您已经学习了更多关于贝叶斯法则的知识，特别是比例关系，现在是解释贝叶斯法则一般形式的时候了。和全概率公式一样，贝叶斯法则可以推广到任意个变量，也可以包含条件变量。根据 9.2 小节中的标记法，您有 3 组变量：X_1，⋯X_n（原因），Y_1，⋯Y_m（“结果”）和 Z_1，⋯Z_l（条件变量）。您将得到原因根据条件变量调节的先验概率 $P(X_1, \cdots X_n \mid Z_1, \cdots Z_l)$，以及同样在条件变量上调节的给定原因下结果条件概率 $P(Y_1, \cdots Y_m \mid X_1,\ldots,X_n, Z_1, \cdots Z_l)$。您希望求出给定结果下原因的概率，同样根据条件变量进行调节。这就是 $P(X_1, \cdots X_n \mid Y_1, \cdots Y_m, Z_1, \cdots Z_l)$。根据贝叶斯法则：

$$P(X_1,\ldots,X_n \mid Y_1,\ldots,Y_m, Z_1,\ldots,Z_l) =$$

$$\frac{P(X_1,\ldots,X_n \mid Z_1,\ldots,Z_l)P(Y_1,\ldots,Y_m \mid X_1,\ldots,X_n,Z_1,\ldots,Z_l)}{\sum_{x_1}\cdots\sum_{x_n} P(X_1 = x_1,\ldots,X_n = x_n \mid Z_1,\ldots,Z_l)P(Y_1,\ldots,Y_m \mid X_1 = x_1,\ldots,X_n = x_n,Z_1,\ldots,Z_l)}$$

我曾经承诺过，在本节中将简化这个公式的标记法。因为分母是规格化因子，您可以使用我们的“成正比”简写形式使这个方程更容易理解：

$$P(X_1,\ldots,X_n \mid Y_1,\ldots,Y_m, Z_1,\ldots,Z_l) \propto P(X_1,\ldots,X_n \mid Z_1,\ldots,Z_l)P(Y_1,\ldots,Y_m \mid X_1,\ldots,X_n, Z_1,\ldots,Z_l)$$

这里的后验概率是多个原因变量上的联合分布，似然率也考虑多个结果变量，而且其他变量（Z 变量）影响着原因和结果，除此之外，这和我们的“后验概率∝先验概率 × 似然率”公式类似。

最后，您可以使用粗体字母 $\mathbf{X}$、$\mathbf{Y}$ 和 $\mathbf{Z}$ 表示变量组。贝叶斯法则可以总结为一个简洁的公式：

$$P(\mathbf{X} \mid \mathbf{Y},\mathbf{Z}) \propto P(\mathbf{X} \mid \mathbf{Z})P(\mathbf{Y} \mid \mathbf{X},\mathbf{Z})$$

其中 $\mathbf{X}$ 表示所有原因变量，$\mathbf{Y}$ 表示所有结果变量，$\mathbf{Z}$ 表示所有条件变量。这个精练的公式是记忆贝叶斯法则一般形式的最佳手段。

现在，您已经学到了估算 Bias 的方法。下一步是用它预测 Toss_{101}。

9.4.2 预测下一次掷币结果

好了，您已经得到了以β分布形式表示的 Bias 后验分布。如何预测下一次掷币的结果？有三种常用方法可以实现这一目标，对于β-二项式模型来说都很简单。如前所述，这几种方法是：

- 最大后验（MAP）方法
- 最大似然估计（MLE）方法
- 全贝叶斯方法

下面按照顺序介绍这些方法。

使用最大后验方法

第一种方法称为**最大后验（MAP）估计**，计算具有最高后验概率密度的 Bias 值。这个值称为 Bias 的**最可能值**，它能使先验概率和似然率的乘积最大。然后，您使用这个 Bias 值预测下一次掷币的结果。

图 9-11 描述了 MAP 过程。第一步是使用前一小节的方法计算 Bias 上的后验分布。从先验分布 beta(2,5)开始，观察到 63 次正面和 37 次背面，得到后验分布 beta(65, 42)。在下一步中，计算 beta(65, 42)出现波峰时的 Bias 值。回顾图 9-9，这就是 beta(65, 42)值最高时的 *x*-轴坐标。换言之，您要得到的是 beta(65, 42)的模，有一个简单的公式可以求得该值：

$$\text{mode}(\text{beta}(\alpha,\beta))=\frac{\alpha-1}{\alpha+\beta-2}$$

在我们的例子中，模等于（65-1）/(65+42-2)，大约等于 0.6095。现在，假定 Bias 等于 0.6095，计算给定数据（63 次正面，37 次背面）的情况下，Toss_{101} 为正面的概率。Toss_{101} 的函数形式说明，掷币结果为正面的概率等于 Bias 的值，即您假定的 0.6095。所以，答案就是 0.6095。

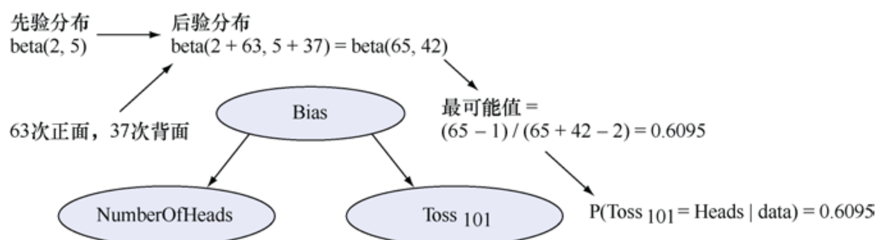


图 9-11　用 MAP 方法预测下一次掷币结果

使用最大似然估计

第二种方法是 MAP 估计过程的常用特例——**最大似然估计**（MLE）。在 MLE 中，您选择“最适合数据”的参数值，而不考虑任何先验分布。MLE 方法有时候被认为不是贝叶斯方法，但是如果假定 Bias 的每个可能值有相同的先验概率，那么它仍然符合贝叶斯框架。这样，公式

后验概率 $\propto$ 先验概率 × 似然率

被简化为

后验概率 $\propto$ 似然率

因此，后验分布的最可能值是使似然率最大的值，**最大似然估计**由此得名。

图 9-12 说明了最大似然方法。除了从先验分布 beta（1,1）开始，为 0～1 的值指定相同概率密度之外，这种方法与图 9-11 中的 MAP 方法类似。如果您记得先验分布的参数是想象的已知正面和背面次数加 1，就能看出这一先验分布表示您没有想象看到任何正面和背面结果的情况。然后，通过相同的计算序列，得出预测值 0.63。这个结果不是巧合，您观察到在 100 次掷币中有 63 次正面。最符合这些观察结果的 Bias 值就是，任何一次掷币得到正面结果的可能性为 0.63。所以，您可以看到最大似然估计选择了最符合数据的参数，而 MAP 估计用先验分布平衡数据。

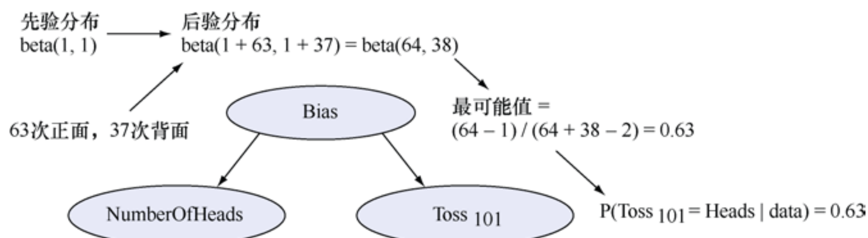


图 9-12　用最大似然方法预测下一次掷币结果

使用全贝叶斯方法

预测下一次掷币结果的第三种方法有时称作**全贝叶斯方法**，因为它不预测单个 Bias 值，而是使用 Bias 的整个后验分布。方法的过程如图 9-13 所示。最初的步骤和 MAP 方法一样，计算 Bias 的后验分布。为了使用这个分布预测 $Toss_{101}$，使用图中所示的公式计算 $P(Toss_{101} = Heads \mid Data)$。这个公式通过全概率公式和链式法则得来。需要注意的是，它包含积分，因为 Bias 是一个连续变量。正如估算后验参数值那样，这个积分可能很难处理，但是β-二项式模型同样很简单。结果是，如果后验分布是 $beta(\alpha_1, \beta_1)$，下一次掷币结果为正面的概率是

$$\frac{\alpha_1}{\alpha_1 + \beta_1}$$

所以，在我们的例子中，正面的概率为 65 / (65 + 42) = 0.6075。为了完成闭环，这个简单的概率公式正是计算β分布参数时在正面和背面次数上加 1 的原因：这样您最终可以得到下一次掷币出现正面概率的简单公式。

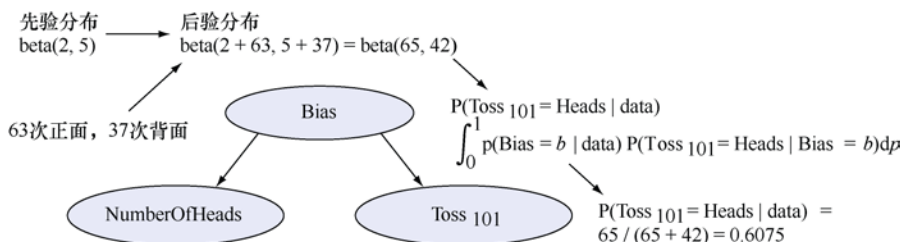


图 9-13 用全贝叶斯方法预测下一次掷币结果

各种方法的对比

了解了这三种方法，下面我们对比它们。

- **MLE 方法**提供对数据的最佳拟合，但是也容易造成过度拟合。过度拟合是机器学习中的一个问题，学习者过于紧密地拟合数据中找到的模式，以至于无法找出显著的特点。在掷币次数较少的情况下，这特别成问题。例如，如果只有 10 次掷币，其中 7 次得到正面结果，您应该立即得出偏差为 0.7 的结论吗？即使是公平硬币投掷 10 次出现 7 次正面的情况也有一定的占比，所以这样的掷币次数不能提供偏币的决定性证据。

  MLE 方法有两个优势，这是它相当流行的原因。首先，它相对高效，因为不需要对所有参数值求积分就能预测下一个实例。其次，它不要求指定一个先验分布，当您没有任何基础时，指定分布是很困难的。但是，对过度拟合的敏感性可能是这种方法的显著问题。

- **MAP 方法**可能是一个好的妥协。包含一个先验可以起到两个作用，一是编码您所拥有的先验信念，二是抵消过度拟合。例如，如果从先验分布 beta(11, 11)开始，就不会使结果以任何方式偏向正面或者背面，但是数据的影响将因为结果中增加了 10 次想象的正面和背面结果而受到抑制。为了说明这一点，假定掷币 10 次得到 7 次正面的结果。先验分布 beta(11, 11)意味着您已经看到了 10 次想象中的正面和 10 次想象中的背面。增加 7 次正面结果和 3 次背面结果，正面和背面次数的总和变为 17 和 13。所以，对偏差的 MAP 估计为 17 / (17 + 13) = 17/30≈0.5667。您也可以从前面给出的β分布求模公式中看出这一点，该公式为

$$\frac{\alpha-1}{\alpha+\beta-2}$$

  对于 7 次正面和 3 次背面的结果，后验分布为 beta(18, 14)，模为 17/30。即使数据中有 70%都是正面结果，您对正面的后验信念仍然只是稍大于 0.5，远远小于 MLE 方法的 0.7。除了可以抵消过度拟合之外，MAP 方法也相对高效，因为它不需要求得所有参数值上的积分。但是它确实需要指定一个先验分布，这可能很困难。
- **全贝叶斯方法**如果可行，可能优胜于其他方法，因为它使用了整个分布。特别是，当分布的模对整个分布不具代表性时，其他方法可能造成误导。对于β分布，这不是严重的问题；MAP 和全贝叶斯预测在我们的例子中可能相互很接近。具体说，从先验分布 beta(11, 11)和观察到的 7 次正面和 3 次背面结果，可以得到后验分布 beta(18, 14)。贝叶斯估算的下一次掷币正面概率为 18 / (18 + 14) = 18/32 = 0.5625，仅仅稍低于 MAP 估算的值。但是，对于其他分布，尤其是具有多个峰值的分布，全贝叶斯方法可能产生比 MAP 方法好得多的估算结果。即使 MAP 方法使用先验分布，也将停留在其中一个峰值上，完全忽略分布的一个重要部分。但是贝叶斯方法从计算上更难以执行。

概率编程系统所支持的方法范围各不相同。最典型的系统支持全贝叶斯推理。因为全贝叶斯推理常常需要积分，这些系统使用逼近算法。有些概率编程系统也支持用于特定模型的最大似然和 MAP 估计，这些方法的计算效率可能更高。特别是，Figaro 提供全贝叶斯和 MAP 算法。第 12 章介绍如何在 Figaro 中实际使用这些方法。

现在您已经知道了推理的基本原则，并理解了贝叶斯推理是如何运用贝叶斯法则从数据中学习，将得到的知识用于未来的预测中。在接下来的几章中，您将学习具体的推理算法。概率编程主要使用两个推理算法族：因子分解算法和抽样算法。这两类算法是下两章的主题。

9.5 小结

- 链式法则使您可以取得单独变量的条件概率分布，构造所有变量上的联合概率模型。
- 全概率公式使您可以取得一组变量上的联合概率分布，对其进行归纳以得到单独变量上的概率分布。
- 概率模型中的网络箭头通常遵循数据生成的过程，而模型中的推理可以采取任何方向。贝叶斯法则帮助您实现这一目标。
- 贝叶斯建模使用贝叶斯法则从结果的观察中推断原因，并使用这些推理预测未来的结果。
- 在贝叶斯推理中，某个变量值的后验概率与该值的先验概率和似然率的乘积成正比，后者是给定该值的情况下得到该证据的概率。
- 在 MAP 估计方法中，使用参数的最可能后验值预测未来实例。
- 在 MLE 方法中忽略先验分布，用最大化似然率的参数值进行预测。这是最简单的方法，但是可能造成数据的过度拟合。
- 在全贝叶斯方法中，使用参数值上的整个后验概率分布预测未来实例。这是最精确的方法，但是在计算上很困难。

9.6 练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．考虑打印机诊断示例的详细打印机模型，见第 5 章图 5-11 中的贝叶斯网络：

- Printer Power Button On = true
- Toner Level = low
- Toner Low Indicator On = false
- Paper Flow = smooth
- Paper Jam Indicator On = false
- Printer State = poor

a）用完整的链式法则写出这种情况的概率，每个变量根据所有前面变量调节。

b）考虑网络中的独立关系，简化这一表达式。

c）编写适用于所有情况的联合概率分布表达式，不指定具体的变量值。

2．对于练习 1 中的网络：

a）编写 Printer Power Button On = true 的概率表达式。

b）编写 Printer Power Button On = true 且 Printer State = poor 的概率表达式。

3．假定 4000 万美国公民中的 1 个将成为美国总统。

a）假定 50%的总统是左撇子，而左撇子在总人口中占 10%。在已知某人是左撇子的情况下，他（或她）成为美国总统的概率有多大？

b）现在假定 15%的美国总统曾就读于哈佛大学，而在总人口中有 1/2000 的人曾就读于哈佛大学。在已知某人曾就读于哈佛大学的情况下，他（或她）成为美国总统的概率有多大？

c）假定在已知某人成为总统的情况下，左撇子和就读于哈佛条件独立，在已知他（或她）是左撇子且曾就读于哈佛的情况下，成为美国总统的概率有多大？

第 10 章　因子分解推理算法

本章介绍如下内容：

- 因子分解推理基础知识，因子的定义和因子上的运算
- 变量消除算法
- 置信传播算法

现在，您已经理解了概率推理的基本原则，下面两章将学习概率编程中使用的一些推理算法。这将帮助您更好地深入理解特定问题的最佳算法，以及设计与算法相适应的模型的方法。

推理算法有两种主要类型。

- **因子分解算法**在称为**因子**的数据结构上运行，因子捕捉用于推理的概率模型。
- **抽样算法**从概率分布中创建可能世界的示例，并用这些例子回答查询。

抽样算法将在下一章中介绍。本章通过如下知识的学习，介绍因子分解算法。

- 因子数据结构和它表现概率模型及查询的方式。您将看到，这和前一章学习的链式法则和全概率公式紧密相连。
- **变量消除（variable elimination，VE）算法**。这是一种**精确算法**，也就是说，它根据模型定义的证据，计算查询的准确概率。因为它是精确的，如果模型支持，它非常有效，但是执行速度可能很慢。

- **置信传播（belief propagation，BP）算法**。这通常是一种**近似算法**。它可能很快，大部分情况下能够返回接近正确的答案，但是并不总是如此。
- 精确和近似算法之间精确度和速度的平衡，以及设计概率模型以适应您所使用算法类型的方法。

为了解释本章中的概念，我从第 5 章中提取了许多关于贝叶斯网络和马尔科夫网络的素材。第 5 章中使用的 Figaro 建模技术相对简单，所以可以随意复习，然后再通读第 6～8 章。您还应该熟悉第 9 章中的推理规则，特别是链式法则和全概率公式。

10.1　因子

对概率分布进行**因子分解（factoring）**的想法类似于整数因子分解。如果您有一个整数，如 15，可以将其分解为 3×5，因此 3 和 5 是 15 的因子。同样，您可以将概率分布分解为其组成因子。在本节中，您将首先学习这些因子的定义。然后，您将了解使用链式法则将概率分布分解成因子的方法。最后，您将看到如何用全概率公式表达包含因子的查询答案。

10.1.1　什么是因子

首先，我将给出因子的通用定义，然后解释它融入概率模型的方式。**因子**是从一组变量值到一个实数的函数的一种表现形式。尽管因子可以有不同的表现形式，但在本书中用一个表格表示。

表 10-1 展示了两个变量（Subject 和 Size）上的一个因子。该因子对于每个变量有一列。因子的右侧还有一列，包含实数值。因子的每一行对应于变量值的特定组合。例如，第一行对应于 Subject = People 和 Size = Small。因子为这些值指定实数 0.25。

表 10-1　两个变量（Subject 和 Size）上的因子。每行对应于变量的特定值，并为这些值指定一个实数

Subject	Size	
People	Small	0.25
People	Medium	0.25
People	Large	0.5
Landscape	Small	0.25
Landscape	Medium	0.5
Landscape	Large	0.25

因子在概率推理中有什么用处？我们回到基本原则。

- 概率分布为每个可能世界指定一个 0 和 1 之间的数。

■ 在一组变量定义的概率模型中，可能世界由每个变量的一个值组成。

概率分布是一个函数，根据模型中每个变量的值，指定 0 和 1 之间的一个数。这意味着，分布可以用一个因子表示。因此，您可以将概率分布看成一个表格，每行表示变量的一个可能赋值。我们将以前几章中的画作场景作为例子，该场景包含变量 Subject、Size 和 Brightness。为了帮助您回忆，图 10-1 中展示了这个例子的贝叶斯网络和条件概率分布（CPD）。

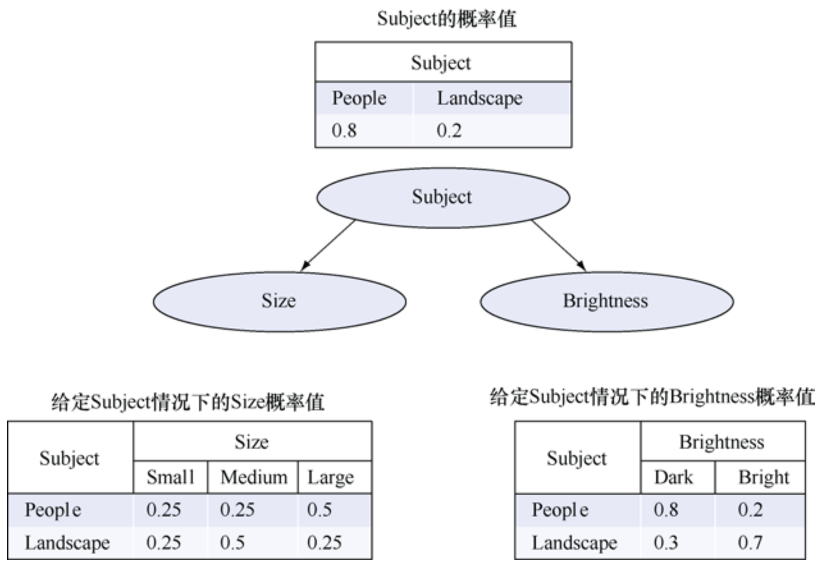


图 10-1 第 5 章中 Subject-Size-Brightness 贝叶斯网络的重现

正如表 10-1，这个网络中的每个 CPD 都可以用一个因子表示。实际上，表 10-1 表示 P(Size | Subject)，也就是图 10-1 中给定 Subject 情况下 Size 的 CPD。例如，根据 CPD，P(Size = Small | Subject = People)为 0.25。相应地，在表 10-1 中，与 Subject = People, Size = Small 行相关的数值为 0.25。与此同时，表 10-2 展示了与 P(Subject)对应的因子。因为 Subject 没有父变量，所以只有对应于 Subject 变量的一列。最后，表 10-3 展示了对应于 P(Brightness | Subject)的因子，该表提及了变量 Brightness 和 Subject。

表 10-2 对应于 P(Subject)的因子

Subject	
People	0.8
Landscape	0.2

表 10-3　对应于 P(Brightness | Subject)的因子

Subject	Brightness	
People	Dark	0.8
People	Bright	0.2
Landscape	Dark	0.3
Landscape	Bright	0.7

当您考虑表示和使用因子的代价时，因子中的行数是重要的因素。您可以使用一个简单公式求得因子中的行数：将因子中每个变量值的数量相乘。例如，表 10-2 对于 Subject 的每个可能值有一行，所以共有 2 行。在表 10-1 中，Subject 和 Size 值的每个组合对应一行，这样的组合有 2×3 个，所以共有 6 行。

注意： 虽然目前您所看到的因子都是 CPD，但是因子是比 CPD 更通用的概念。在变量消除等算法的运行过程中会创建各种不与 CPD 对应的因子。它们可以表示任何函数，从变量值到实数，不管这些函数的来源是什么。此外，虽然因子代表的函数值总是实数，但是它不一定是概率。实际上，因子代表的值不一定在 0 和 1 之间。例如，在马尔科夫网络中，可以从势中得出因子，这些数值可能大于 1。

现在您已经理解了因子的概念，但是它与概率分布的分解有何关系？

10.1.2　用链式法则分解概率分布

理解因子和因子分解算法的第一个关键是链式法则，我们在第 9 章中学习了这个法则。链式法则使您可以用条件概率的乘积计算许多变量的某个赋值的联合概率。链式法则与贝叶斯网络紧密相关，对贝叶斯网络定义的概率分布不可或缺。

例如，考虑两个变量 Subject 和 Brightness。链式法则说明，可以用如下公式计算“这幅画作是鲜艳的人物画”的概率：

P(Subject = People, Brightness = Bright) =

P(Subject = People) P(Brightness = Bright | Subject = People) =

$0.8 \times 0.2 = 0.16$

数字 0.8 和 0.2 是从哪里得到的？您可以在因子中找到它们。确切地说，0.8 来自于表 10-2 中，Subject 取值为 People 那一行中的 P(Subject)因子。类似地，0.2 来自于表 10-3 中 Subject 取值为 People、Brightness 取值为 Bright 那一行中的 P(Brightness | Subject)因子。

您可以从两个因子的对应行中获得数值，对 Subject 和 Brightness 的所有值进行类似的计算。图 10-2 展示了两个因子相乘的结果，这称作**因子乘积**。因子乘积的概念很重要——将变量值相等的行相关的数值相乘。但是有些变量只出现在一个或者另一个因子

中，而不会在两者中都出现。结果因子中的行将提及（a）同时出现在两个因子中的所有变量、（b）只出现在第一个因子中的所有变量和（c）只出现在第二个因子中的所有变量。为了取得与结果因子中的一行相关的数字，您可以找出第一个因子中有相同（a）和（b）变量的行，以及第二个因子中有相同（a）和（c）变量的行。然后，将与这两行相关的数值相乘。我们用 P(Subject) 和 P(Brightness | Subject)的因子详细解释。要将这两个因子相乘，采用如下的步骤。

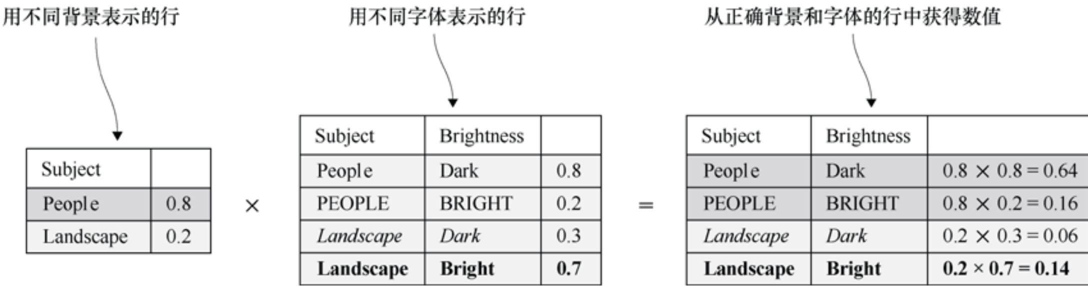


图 10-2 因子乘积。各行进行编码以说明结果因子中数字的来源。例如，结果因子中(People, Bright)一行来自第一个输入中的 People（深色背景）行和第二个输入中的(People, Bright)（全大写）行

1．创建一个新因子，其变量是出现在两个输入中任何一个的所有变量。这个因子中的行对应于这些变量值的所有组合。在我们的例子中，变量是 Subject 和 Brightness，因此有对应于(People, Dark)、(People, Bright)、(Landscape, Dark)和(Landscape, Bright)的行。

对于结果因子中的每一行——如（People, Bright）——找出第一个输入中与此行一致的行，得到该行中的数字。在我们的例子中，结果行中的 Subject 值是 People，所以我们必须找到输入中 Subject = People 的行。图中进行了编码，输入中的 Subject = People 行采用深色背景，所以结果因子中的(People, Bright)行也采用深色背景。您得到了数值 0.8。

2．对第 2 个因子进行同样操作：找出与结果因子中一致的行，并得到该行中的数字。在这个例子中，第二个因子的(People, Bright)行使用全大写字体，您得到数值 0.2。您已经知道如何将两个因子相乘，可以求取任意多个因子的乘积。在第 9 章中您已经了解到，贝叶斯网络定义的概率分布可以用链式法则表达。为了保持公式的简短，我将使用如下简写。

- Subject 简写为 J。
- Size 简写为 Z。
- Brightness 简写为 B。

在我们的例子中，可以将公式写为：

$P(J, Z, B) = P(J)\ P(Z \mid J)\ P(B \mid J)$

上式没有指定 Subject、Size 和 Brightness 的特定值，适用于这些变量的所有值。因为 P(Subject)、P(Size | Subject)和 P(Brightness | Subject)都是因子，上式说明 Subject、Size 和 Brightness 的联合概率分布可以用 3 个因子的乘积表示，如表 10-4。

表 10-4　Subject、Size 和 Brightness 之上的联合概率分布可以通过 3 个因子的乘积求得，每一行都从输入因子中的对应行得到数字

Subject	Size	Brightness	
People	Small	Dark	$0.8 \times 0.25 \times 0.8 = 0.16$
People	Small	Bright	$0.8 \times 0.25 \times 0.2 = 0.04$
People	Medium	Dark	$0.8 \times 0.25 \times 0.8 = 0.16$
People	Medium	Bright	$0.8 \times 0.25 \times 0.2 = 0.04$
People	Large	Dark	$0.8 \times 0.5 \times 0.8 = 0.32$
People	Large	Bright	$0.8 \times 0.5 \times 0.2 = 0.08$
Landscape	Small	Dark	$0.2 \times 0.25 \times 0.3 = 0.015$
Landscape	Small	Bright	$0.2 \times 0.25 \times 0.7 = 0.035$
Landscape	Medium	Dark	$0.2 \times 0.5 \times 0.3 = 0.03$
Landscape	Medium	Bright	$0.2 \times 0.5 \times 0.7 = 0.07$
Landscape	Large	Dark	$0.2 \times 0.25 \times 0.3 = 0.015$
Landscape	Large	Bright	$0.2 \times 0.25 \times 0.7 = 0.035$

您已经知道，因子中的行数是每个变量值数量的乘积。所以在表示联合分布的这个因子中，Subject 有 2 个值，Size 有 3 个值，Brightness 有 2 个值，总行数为 $2 \times 3 \times 2 = 12$。一般来说，行数与变量数量成指数关系。这也就意味着，当您处理的不是小型网络时，代表联合分布的因子会变得过大，无法表示，也无法用其进行推理。

当您取得一个整数，并用较小整数的乘积表示它时，这种运算就是整数的因子分解。概率分布也是如此。而且，正如整数，用比整数小得多的因子进行乘除可能很容易；概率分布也是如此。用一小组变量上的因子进行概率推理比使用与变量数成指数关系的整个联合分布更容易。这也就解释了因子对概率推理的重要性。

10.1.3　使用全概率公式，定义包含因子的查询

到目前为止，您已经了解如何以因子乘积的形式表达联合概率分布。下一步是用因子表达查询的答案，这一步的关键是全概率公式。

没有证据的查询

首先，我们假定没有证据。您对得到特定变量（如 Brightness）的概率分布感兴趣。首先从联合概率分布入手；在我们的例子中，分布如表 10-4 所示。全概率公式告诉我们，如果您对 Brightness 取某个特定值（如 Bright）的概率感兴趣，可以将 Brightness 取得该值的联合概率分布累加起来。这个分布如表 10-5 所示，该表中的概率分布与表

10-4 相似，但是所有 Brightness = Bright 的行都用粗体显示。由此可以得出 Brightness = Bright 的概率是所有 Brightness 取值为 Bright 的行中数字的总和，也就是所有以粗体显示的行。您可以得到如下的结果：

$$P(Brightness = Bright) = 0.04 + 0.08 + 0.04 + 0.035 + 0.035 + 0.07 = 0.3$$

表 10-5 Subject、Size 和 Brightness 的联合概率分布，对应于 Brightness = Bright 的所有行以粗体显示

Subject	Size	Brightness	
People	Small	Dark	0.16
People	**Small**	**Bright**	**0.04**
People	Medium	Dark	0.16
People	**Medium**	**Bright**	**0.04**
People	Large	Dark	0.32
People	**Large**	**Bright**	**0.08**
Landscape	Small	Dark	0.015
Landscape	**Small**	**Bright**	**0.035**
Landscape	Medium	Dark	0.03
Landscape	**Medium**	**Bright**	**0.07**
Landscape	Large	Dark	0.015
Landscape	**Large**	**Bright**	**0.035**

同样，可以将 Brightness 取值为 Dark 的所有行（浅色字体）中的数字加总得到 Brightness = Dark 的概率。将两个数字放在一起，就可以得到表 10-6 所示的因子。

表 10-6 表示 Brightness 概率分布的因子。每一行是联合因子中对应行数字的总和。Brightness = Bright 行用粗体显示，表示其中的数字从表 10-5 用粗体显示的行中取得

Brightness	
Dark	0.16 + 0.32 + 0.16 + 0.015 + 0.015 + 0.03 = 0.7
Bright	**0.04 + 0.08 + 0.04 + 0.035 + 0.035 + 0.07 = 0.3**

您刚刚进行的运算有两个常用名称。一是**因子总和**。不要混淆：总和并不意味着把两个因子相加，而是将单一因子各行的数字加总得出一个更简单的新因子。当您进行加总运算时，就从结果因子中删除了某些变量；在我们的例子中，您删除了 Subject 和 Size。您可以说这是**对这些变量求和**。有时候您还会看到另一个名称——**边缘化**。Brightness 上的结果概率分布称作边缘概率，与开始时的联合分布截然相反。我将避免使用边缘化这个术语，但是因子总和的概念在分解推理中很重要。

您可以编写一个数学公式表达这一概念。因子总和的数学标记是希腊字母Σ。要表示从 P(Subject, Size, Brightness)中加总 Subject 和 Brightness 的概率，可以写出如下公式：

$$P(B) = \Sigma_{J,Z}\ P(J, Z, B)$$

在前一小节中得知

$$P(J, Z, B) = P(J)\ P(Z \mid J)\ P(B \mid J)$$

组合上面两个公式，得到：

$$P(B) = \Sigma_{J,Z}\ P(J)\ P(Z \mid J)\ P(B \mid J)$$

您已经使用因子的乘积和加总运算表达查询的答案。结果表达式被称作**乘积之和表达式**。最终因子中的数字是从输入因子中数字相乘得到的数字的总和。因子分解推理算法操纵乘积之和表达式回答查询。

有证据的查询

如果有证据，会发生什么情况呢？假定您已经观察到 Brightness = Bright，希望查询 Subject 的后验概率。换言之，您想要计算 P(Subject | Brightness = Bright)。处理证据的最简单方法就是引入新的因子，编码您的证据。

正如第 4 章中所述，根据证据的调节是通过排除与证据不符的所有可能世界（将其概率设置为 0）进行的。这可以通过创建一个因子，为与证据不符的状态指定 0 值实现。在我们的例子中，要编码证据 Brightness = Bright，可以引入表 10-7 中的因子。

表 10-7　编码证据 Brightness = Bright 的因子

Brightness	
Dark	0
Bright	1

这个因子有何效果？当您将这个因子乘以任何其他因子，Brightness 取值为 Dark 的行将自动得到数字 0，而其他行不受影响。在联合分布中，您将有效地删除所有 Brightness 取值为 Dark 的行，其他可能世界则保持不变。

如果将这个因子称作 E_B，就得到如下的乘积之和表达式：

$$P(J, B = Bright) = \Sigma_{Z,B}\ P(J)\ P(B \mid J)\ P(Z \mid J)\ E_B(B)$$

表 10-8 展示了这个概率分布，它是 P(Subject)、 P(Brightness | Subject)、 P(Size | Subject)的乘积再乘以证据因子 E(Brightness = Bright)。注意，所有与证据不一致的行中的条目为 0；那些行将被“删去”。

下一步是执行加总运算。您对没有出现在查询中的变量（Size 和 Brightness）求和。对于 Subject 的每个值，将表 10-8 中所有非零行相加。结果是表 10-9 中的因子，它代表 P(Subject, Brightness = Bright)。

表 10-8　Subject、Size 和 Brightness 的联合概率分布乘以 Brightness = Bright 的证据因子

Subject	Size	Brightness	
People	Small	Dark	0.16 × 0 = 0
People	Small	Bright	0.04 × 1 = 0.04
People	Medium	Dark	0.16 × 0 = 0
People	Medium	Bright	0.04 × 1 = 0.04
People	Large	Dark	0.32 × 0 = 0
People	Large	Bright	0.08 × 1 = 0.08
Landscape	Small	Dark	0.015 × 0 = 0
Landscape	Small	Bright	0.035 × 1 = 0.035
Landscape	Medium	Dark	0.03 × 0 = 0
Landscape	Medium	Bright	0.07 × 1 = 0.07
Landscape	Large	Dark	0.015 × 0 = 0
Landscape	Large	Bright	0.035 × 1 = 0.035

表 10-9　对表 10-8 中的 Size 和 Brightness 因子求和

Subject	
People	0.04 + 0.08 + 0.04 = 0.16
Landscape	0.035 + 0.035 + 0.07 = 0.14

如果仔细观察前面的因子，就会发现它不能精确地回答您的查询。您希望得到条件概率 P(Subject | Brightness = Bright)；例如，您想知道在已知画作颜色鲜艳的情况下，画作是人物画的概率。但是前面的因子只提供 P(Subject, Brightness = Bright)，这能够告诉您画作是人物画且颜色鲜艳的概率。

上述问题很容易解决。概率分布 P(Subject, Brightness = Bright)是查询的未规格化答案。您可以发现，表 10-9 中的因子总和不为 1，它们之间的比例为 16∶14。为了得到查询的答案，可以规格化这个因子，使数字加总为 1 同时保持比例，从而得到表 10-10 的最终结果。

表 10-10　规格化表 10-9 中的结果所得到的 P(Subject | Brightness = Bright)

Subject	
People	0.16 / (0.16 + 0.14) = 0.5333
Landscape	0.14 / (0.16 + 0.14) = 0.4667

现在，您已经了解什么是因子，学习了因子上的乘积和加总运算，并且知道如何以乘积之和表达式表示查询的答案。现在，您似乎必须将所有因子相乘，创建联合分布以回答查询。但是整个联合分布的大小和模型中的变量数量成指数关系。因子分解推理算法的主要目标是避免创建整个分布而代之以更紧凑的形式，这是下一节所要学习的内容。

10.2 变量消除算法

在前一小节中，您已经看到查询的答案可以由因子上的乘积之和表达式定义。**变量消除**（VE）法是一种操纵上述表达式的算法。基本运算很简单：使用简单的代数运算，每次从表达式中消除一个非查询变量——这就是**变量消除**法名称的来源。因为 VE 仅使用代数规则操纵表达式，所以它是一种精确推理算法。

尽管从根本上说变量消除法是一种基于代数的算法，但是它也可以通过图形理解。图形视图直观地说明了变量消除过程，有助于理解这种算法的复杂性。代数视图对理解算法的运行细节很重要。相应地，我将首先提供图形视图，帮助您对这种算法有总体上的理解，然后介绍详细的代数视图。

10.2.1 VE 的图形解释

VE 的图形解释很直观，只需要几句话和一张图。在 VE 中，您有一个或者多个查询变量，按照某一顺序消去查询变量之外的所有变量。当您消去一个变量时，取得提及该变量的因子，生成一个新因子。创建这些图的目的是跟踪计算中的任何一个时点，同一因子中出现了哪些变量。

图 10-3 展示了一个 4 节点贝叶斯网络，包含变量 Subject、Size、 Brightness 和 Material。查询变量为 Size，您将按照 Material、Subject 和 Brightness 的顺序消去变量，这是一个随意选择的顺序。您可以使用任何消去顺序，但是之后您会看到，这会造成一定的差异。

图 10-3 展示了如下的步骤。

1．第 1 步只是复制原始贝叶斯网络。

2．根据原始网络，开始构建初始 VE 图。对于网络中的每个节点，这个图中也包含一个对应的节点，在同一个因子提及的任意两个节点之间有一条无向边。这包括贝叶斯网络中每个父变量和子变量之间的边，以及同一子变量的任意两个父变量之间的边。例如，Subject 和 Material 都是 Brightness 的父变量，因为两者都被编码 Brightness CPD 的因子提及，所以两者间存在一条边。

3．一个接一个地消去变量，在消去一个变量时，在连接到被消去变量的任意两变量之间增加一条边（如果它们尚未连接的话）。这两个变量都被消去变量所创建的因子所提及。因为它们出现在同一个因子中，所以必须连接。

在我们的例子中，Material 首先被消去。因为 Subject 和 Brightness 已经相连，所以不需要添加新的边。

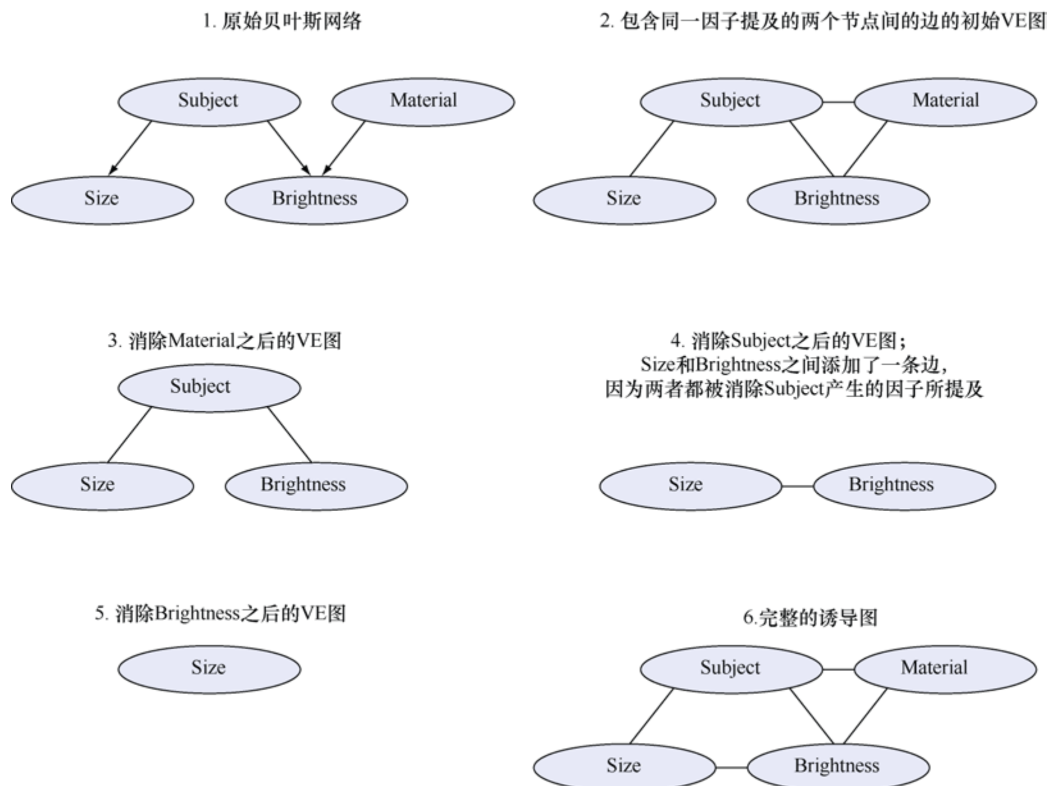


图 10-3 通过变量消除过程构建一个 VE 图

4．接下来消去 Subject。因为 Size 和 Brightness 都连接到 Subject，它们在结果中相互连接。

5．最后被消去的变量是 Brightness，没有添加新的边，只剩下 Size。

6．最后，将任何图中出现的所有边组合成单一图。这个图称作消去 Material、Subject 和 Brightness 导出的图，或者简单地称作**诱导图**。

术语：从一个贝叶斯网络开始时，初始 VE 图被称作**端正图**，原因是端正图是通过连接相同子节点的双亲使其“联姻”而得到的。

理解算法的复杂性

上述图形解释很有用，因为它帮助您理解了算法的复杂性。VE 涉及许多加法和乘法。加法和乘法的数量与所创建因子的大小紧密相关。诱导图为您提供了操纵因子大小的很好手段，因为它告诉您哪些变量出现在同一个因子中。

这种分析依赖于图论中团的概念。**团**（clique）是图中相互连接的一组节点。如果诱导图中的一组节点组成团，这些节点必然在某个时点一起出现在某个因子中。实际上

不难看出，出现在同一个因子中变量的最大数量就等于诱导图中最大团的顶点数。

例如，在图 10-3 右下角的诱导图中，Subject、Size 和 Brightness 变量都相互连接，所以它们组成了一个团。这些变量都出现在消除 Subject 时产生的中间因子——在乘以提及 Subject 的因子之后，但是在加总结果中的 Subject 之前。类似地，Subject、Material 和 Brightness 也相互连接，且出现在 Brightness 的 CPD 中。但是 Size 和 Material 不相连，所以 Size、Subject、 Brightness 和 Material 没有组成一个团。确实，在任何时候都没有任何因子同时提及 Size 和 Material。

现在，您知道出现在一个因子中的最大变量数量是最大团的顶点数。该因子有多大呢？记住这个公式：要得到因子中的行数，将每个变量取值的数量相乘。结果与变量的数量成指数关系。这导致了如下的关键结果：**在给定变量消除顺序下，VE 的复杂度与变量消除顺序导出的图中最大团的顶点数成指数关系。**

使用不同的变量消除顺序

前面的分析遵循某种给定的变量消除顺序。使用哪一种顺序是否重要？答案是，这会带来很大的差异。图 10-3 中的诱导图有两个顶点数为 3 的团，最大团顶点数为 3。

图 10-4 展示了两种备选顺序产生的诱导图。左侧的诱导图中最大团的顶点数为 4，因为所有变量都相连。这意味着使用这一顺序，VE 的代价将更高。右侧使用的顺序不会导致添加任何新的边。虽然最大团的顶点数和最初的顺序一样，但是只有一个顶点数为 3 的团，所以使用这个顺序的 VE 更为经济。

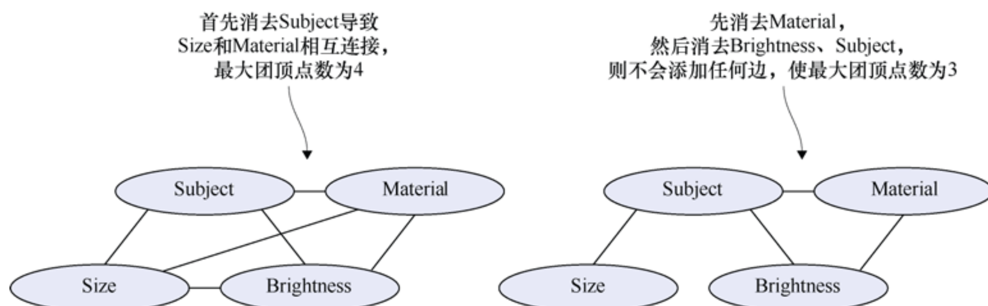


图 10-4　从两种备选变量消除顺序得出的诱导图。左：Subject-Material-Brightness，右：Material-Brightness-Subject

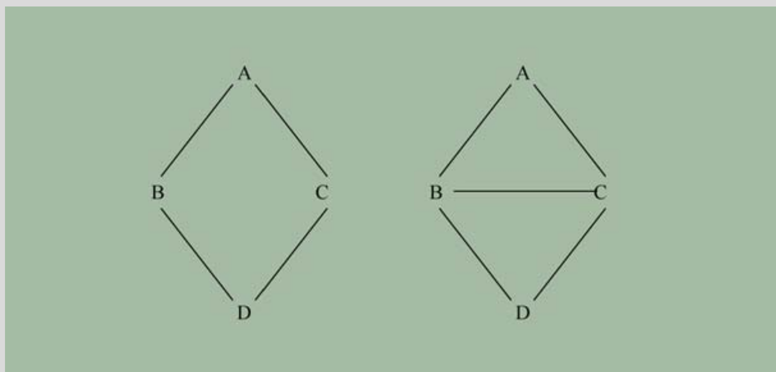
最后一种消去顺序很有趣，因为没有添加任何新边。在添加一条边时，您迫使打算连接的变量出现在同一个因子中。这使得因子更大，推理的代价更高。如果您采用一种不增加任何边的消去顺序，就有可能达到更好的效果，因为在推理过程中产生的因子不会将一开始就共同包含在某个因子中的变量集合起来。

这就自然产生了一个问题：是否总有不增加任何边的消去顺序？这一答案与**三角化**的概念有关。如果所有环都是“三角形”，则称图被“三角化”了；补充材料“三角化

图”解释了这一概念的含义。如果初始 VE 图是三角化图，则至少有一种不增加任何边的消去顺序。相反，如果初始 VE 图不是三角化的，不管使用何种消去顺序，都至少需要增加一条边。确保初始 VE 图是否三角化的一种方法（但不是唯一的方法）是从完全没有环的网络开始。因此，**如果初始 VE 图没有环，总是可以从图的两端向内消去节点，而不需要增加任何边。**

三角化图

如果图中的所有环都由三角形组成，则该图是**三角化**的。从技术上讲，这意味着长度为 4 或更大的图中所有回路的两个不相邻边之间都存在交叉的边。用一张图就很容易理解这一点。



三角化图，左侧的图包含没有交叉边的回路 A-B-D-C，所以它不是三角化图。在右侧的图中，这个回路有交叉的边 B-C，所以它是三角化图。确实，它是由三角形 A-B-C 和 B-C-D 组成的

为什么三角化能够决定能否在不增加边的情况下消除变量？观察左侧的图可以看出，如果首先消去 A 或者 D，就将增加边 B−C，而如果先消去 B 或者 C，就会增加边 A−D。不管怎么做，都会增加边。而在右侧的图中，您可以首先消去 A 而不增加边。此时，剩下的是三角形 B−C−D，其中的顶点都相连，您可以以任何顺序消去变量而不会增加边。

这是一个通用的原则。非三角化图有长度为 4 或者更大、没有交叉边的环。当您第一次消除该环中的某个节点，就会增加一条交叉的边。相反，在三角化图中，总有一个节点，从其开始变量消除不需要增加边。而且，在消去该节点之后，图仍然是三角化的，所以可以继续消除而不增加边。结论是：当且仅当初始 VE 图是三角化图时，存在不增加边的变量消除顺序。

这有何意义？如果图是三角化的，您在推理中产生的因子不会大于开始时使用的因子。所以如果从一开始就能这样表现模型，就应该能够高效地进行推理。反之，如果图不是三角化的，您在推理过程中就可能创建大得多的因子。得出好的顺序。流行的启发式方法之一使用了尽可能避免增加边的直觉。这种启发式方法在每个时点都选择消去时在图中增加最少边的节点。Figaro 使用了这种方法的一个变种，还考虑了所涉变量的取值数量。

消去顺序很重要，如何找到一个好的顺序？遗憾的是，这从计算上说很难，其代价

通常与最佳顺序下最大团顶点数成指数关系。但是，您可以使用启发式方法。

10.2.2　VE 代数运算

本小节详细说明 VE 算法。如果您不想了解所有这些细节，可以跳到最后的算法总结。

为了理解 VE，我们举一个简单的例子。您希望计算 P(Subject, Brightness = Bright)，这个概率由如下的乘积之和表达式定义：

$P(J, B = Bright) = \Sigma_{Z,B}\ P(J)\ P(B \mid J)\ P(Z \mid J)\ E_B(B)$

我们写下 P(J, B = Bright)的因子，并明确每个条目的计算方法。表 10-11 用符号表示了这些计算方法，表 10-12 则展示了具体的数值计算。

表 10-11　由明确产生每个条目得到的 P(Brightness)查询因子。因为证据指定 Brightness = Bright，任何 Brightness = Dark 的条目被设置为 0，所以处理证据因子时可以简单地从计算中删除这些条目

Subject	
People	P(J = People) P(B = Bright | J = People) P(Z = Small | J = People) E_B(B = Bright) + P(J = People) P(B = Bright | J = People) P(Z = Medium | J = People) E_B(B = Bright) + P(J = People) P(B = Bright | J = People) P(Z = Large | J = People) E_B(B = Bright) + P(J = People) P(B = Dark | J = People) P(Z = Small | J = People) E_B(B = Dark) + P(J = People) P(B = Dark | J = People) P(Z = Medium | J = People) E_B(B = Dark) + P(J = People) P(B = Dark | J = People) P(Z = Large | J = People) E_B(B = Dark)
Landscape	P(J = Land.) P(B = Bright | J = Land.) P(Z = Small | J = Land.) E_B(B = Bright) + P(J = Land.) P(B = Bright | J = Land.) P(Z = Medium | J = Land.) E_B(B = Bright) + P(J = Land.) P(B = Bright | J = Land.) P(Z = Large | J = Land.) E_B(B = Bright) + P(J = Landscape) P(B = Dark | J = Land.) P(Z = Small | J = Land.) E_B(B = Dark) + P(J = Landscape) P(B = Dark | J = Land.) P(Z = Medium | J = Land.) E_B(B = Dark) + P(J = Landscape) P(B = Dark | J = Land.) P(Z = Large | J = Land.) E_B(B = Dark)

表 10-12　P(Brightness)因子，用 CPD 得出的数值代替上表中的各项。数字用不同字体编码，表示其来源因子。用小字体表示来自 P(Subject)，粗体表示来自 P(Brightness | Subject)，常规字体表示来自 P(Size | Subject)，斜体表示来自 E_B(Brightness)。这有助于在进行计算时跟踪数字

Subject	
People	0.8 × **0.8** × 0.25 × *1* + 0.8 × **0.8** × 0.25 × *1* + 0.8 ×**0.8** × 0.5 × *1* + 0.8 × **0.2** × 0.25 × *0* + 0.8 × **0.2** × 0.5 × *0* + 0.8 × **0.2**× 0.25 × *0*
Landscape	0.2 ×**0.3** × 0.25 × *1* + 0.2 × **0.3** × 0.25 × *1* + 0.2 × **0.3** × 0.5 × *1* + 0.2 × **0.7** × 0.25 × *0* + 0.2 × **0.7** × 0.5 × *0* + 0.2 × **0.7** × 0.25 × *0*

VE 通过每次消去一个答案中不需要的变量进行这些计算。在我们的例子中，您必须消去变量 Brightness 和 Size。我们首先消去 Brightness。将来自提及 Brightness 的因子中的数字相乘，还要避免与其他数字相乘。

实现这一目标的第一步是移动乘积中的所有数字，使来自提及 Brightness 的因子的数字移到右侧，来自其他因子的数字移到左侧。在我们的例子中，粗体和斜体数字分别来自于 P(Brightness | Subject)和 E_B(Brightness)，所以它们被移到右侧，其他数字移到左侧。这一操作是合理的，因为乘法交换律意味着可以改变乘法中各项的顺序。结果如表 10-13 所示。注意来自于 P(Brightness | Subject)和 E_B(Brightness)的粗体和斜体数字是如何移到乘积的右侧的。

需要注意的一点是，尽管我已经说明本身可以移动的数字，但是表 10-13 提供了因子的乘积之和表达式；正如下式所示，您已经将 P(B | J) 和 E_B(B)移到没有提及 B 的因子右侧。

$$P(J, B = \text{Bright}) = \Sigma_{Z,B}\ P(J)\ P(Z \mid J)\ P(B \mid J)\ E_B(B)$$

下一步是使用分配率：

$$a \times b + a \times c = a \times (b + c)$$

在我们的例子中，这意味着：

$$0.8 \times 0.25 \times 0.8 \times 1 + 0.8 \times 0.25 \times 0.2 \times 0 = 0.8 \times 0.25 \times (0.8 \times 1 + 0.2 \times 0)$$

运用这一法则，可以将表 10-13 中的因子改写为表 10-14。

表 10-13 我们的 P(Brightness)乘积之和表达式，来自提及 Brightness 的因子的数字移到右侧

Subject	
People	0.8 × 0.25 × **0.8** × *1* + 0.8 × 0.25 × **0.8** × *1* + 0.8 × 0.5 ×**0.8** × *1* + 0.8 × 0.25 × **0.2** × *0* + 0.8 × 0.5 × **0.2** × *0* + 0.8 × 0.25 × **0.2** × *0*
Landscape	0.2 × 0.25 × **0.3** × *1* + 0.2 × 0.25 × **0.3** × *1* + 0.2 × 0.5 × **0.3** × *1* + 0.2 × 0.25 × **0.7** × *0* + 0.2 × 0.5 × **0.7** × *0* + 0.2 × 0.25 × **0.7** × *0*

表 10-14 取得表 10-13 中的因子并创建仅涉及提及 Brightness 的因子的内和

Subject	
People	0.8 × 0.25 × (**0.8** × *1* + **0.2** × *0*) + 0.8 × 0.25 × (**0.8** × *1* + **0.2** × *0*) + 0.8 × 0.5 × (**0.8** × *1* + **0.2** × *0*)
Landscape	0.2 × 0.25 × (**0.3** × *1* + **0.7** × *0*) + 0.2 × 0.5 × (**0.3** × *1* + **0.7** × *0*) + 0.2 × 0.25 × (**0.3** × *1* + **0.7** × *0*)

您已经再次进行了因子上的运算，创建了仅包含提及 Size 的因子中的数字的内和。这些数字就是前一步中移到右侧的数字；在我们的例子中，就是常规字体的数字。按照因子的说法，您已经创建了如下乘积之和表达式：

$$P(J, B = \text{Bright}) = \Sigma_Z\ P(J)\ P(Z \mid J)\ \Sigma_B\ P(B \mid J)\ E_B(B)$$

消去 Brightness 的最后一步是计算乘积的内和，换言之，计算Σ_BP(B | J) E_B(B)。计算的结果是一个因子。因为这一计算从提及 Brightness 和 Subject 的因子开始，且加总 Brightness，结果因子仅提及 Subject，该因子如表 10-15 所示。

我们将这个因子命名为 F_B，也就是从消去 Brightness 中得到的因子。您可以将这个因子代入方程式 3 得到如下表达式：

$$P(J) = \Sigma_Z P(J)\ P(Z \mid J)\ F_B(J)$$

瞧瞧，这就是不提及 Brightness 的乘积之和表达式。您已经成功地消去了 Brightness，而且不必将所有因子相乘。

表 10-15　计算 $\Sigma_B P(B \mid J)\ E_B(B)$ 得到的因子。这个因子中的条目以漂亮的字体显示

Subject	
People	**0.8** × *1* + **0.2** × *0* = *0.8*
Landscape	**0.3** × *1* + **0.7** × *0* = *0.3*

算法总结

我已经非常详尽地说明了这个过程，它很简单。下面是从乘积之和表达式 S 中消去变量 V 的算法：

```
Eliminate(V, S) {
  Move all factors in S that mention V to the right
  Move the summation over V so that it encloses only those factors
  Compute the inner sum-of-products involving the factors that mention V
  Replace this sum-of-products in S with the resulting factor
}
```

这就是 VE 的大部分内容，下面是 VE 的基本算法：

```
VariableElimination(S) {
  Choose an elimination order O that contains all variables except the
      query variables
  For each variable V in O {
    Eliminate(V, S)
  }
}
```

10.3　VE 的使用

您已经知道了 VE 的工作原理，下面讨论如何在实践中应用 VE。我将首先介绍在 Figaro 中使用 VE 的机制，以及不同的变种。然后，我将介绍为 VE 设计模型时的一些通用原则，最后描述一些实际应用。

10.3.1　Figaro 特有的 VE 考虑因素

迄今为止，您已经研究了涉及贝叶斯网络的简单示例。概率程序可能远比贝叶斯网络复杂，包含更加丰富的变量集、数据结构、控制流甚至递归。幸运的是，Figaro 全面负责这些复杂工作的处理。

Figaro 中特有的 VE 版本是已经在本书中多次使用的 VariableElimination 算法。复习一下，VE 通过将查询目标列表传递给 VariableElimination 构造程序运行，后者创建一个算法，然后进行查询。例如：

```
val algorithm = VariableElimination(element1, element2)
algorithm.start()
println(algorithm.probability(element1, 0))
println(algorithm.probability(element1, (i: Int) => i > 0))
println(algorithm.distribution(element1).toList)
println(algorithm.mean(element2))
println(algorithm.expectation(element2, (d: Double) => d × d)
```

打印 element1 值为 0 的概率

打印 element1 值大于 0 的概率

打印 element1 的概率分布。algorithm.distribution 返回一个流，所以您将其转换为一个列表以便打印

打印双精度元素 element2 的均值

打印 element2 平方的期望值

您还看到了只需一行的快捷方式：

```
VariableElimination.probability(element, 0)
VariableElimination.probability(element, (i: Int) -> i > 0)
```

Figaro 特有的 VE 算法将概率程序转换成一个巨大的贝叶斯网络。这一转换成功需要如下条件：模型中可能存在的变量数量有限。这排除了一些开放宇宙模型，在这类模型中对象的数量没有上限，同时也排除了递归次数不限的递归模型。

而且，VE 本身是一种离散算法，难以处理连续分布问题。Figaro 对此的解决方案是对模型中出现的每个原子连续元素抽样一组值。因为只使用可能值的一个小子集，VE 在这种情况下不再是精确算法。但是这对您的目的可能已经足够好了，尤其是在模型中没有太多连续元素的情况下。

如果您希望坚持使用标准算法，可以直接跳到下一节。但是 Figaro 团队不断地尝试推动最新型的概率编程推理算法，VE 算法也不例外。如果您愿意进一步挖掘，就可以从这些进步中获益。

Figaro 的 VE 替代方案之一**惰性** VE 放松了对无限递归的限制。对这一算法的完整解释超出了本书的范畴，但是我希望您意识到它的存在。惰性 VE 的主要观念是，您可以部分扩展该模型，在这个部分扩展的模型上运行 VE，计算查询答案的上下界。例如，想象生成一定规模的图，并估算图的某个属性的程序。图的大小可能没有上界，所以无法应用常规版本解决这个问题。但是惰性 VE 可以部分扩展该模型，生成达到某一上界的图，并用部分扩展估算待估算属性的概率分布。

Figaro 团队还致力于新的分解推理通用框架——**结构化分解推理**。这个框架包含一个结构化 VE 算法，该算法在一个试验性的软件包，很快将移入主软件包中。（到您阅读本书时，它可能已经在那里了）结构化 VE 的思路是使用概率程序的结构指导解题过程。每当链扩展的时候，创建一个子问题表示链函数创建的所有元素。这个子问题内的所有变量可以通过在子问题上运行的单独 VE 过程消去。这种方法的主要好处之一是，

如果同一个子问题在问题中出现多次，第一次的解决方案可以重用。

10.3.2　设计模型支持高效的 VE

现在，您已经理解了 VE 及其复杂性，如何在设计模型时运用这些知识？本节提供一些技巧。

避免过多的闭环

最显而易见的一点是尽可能避免在模型中出现没有交叉边的环。如果您有一个没有交叉边的环，最终就需要增加一条边，这将增大推理的代价。您已经看到，如果有一个三角化的网络，每个长度为 4 或更大的环都有交叉边，就可以在不增加任何边的情况下执行 VE。

如果无法完全避免环，孤立的环比许多紧密相连的环更好。图 10-5 展示了 VE 难以解决的网络的一个示例。这是您在第 5 章中见到过的图像恢复马尔科夫网络。图中展示了一个 4×4 网络，但是您可能需要解决数百甚至数千个像素。

遗憾的是，不管选择何种变量消除顺序，都没有办法避免增加边，创建顶点数量为图像短边像素数的团。VE 的代价与网格的大小成指数关系。因此，尽管 VE 从原理上适合于此类网络，在实践中需要花费极长的时间。

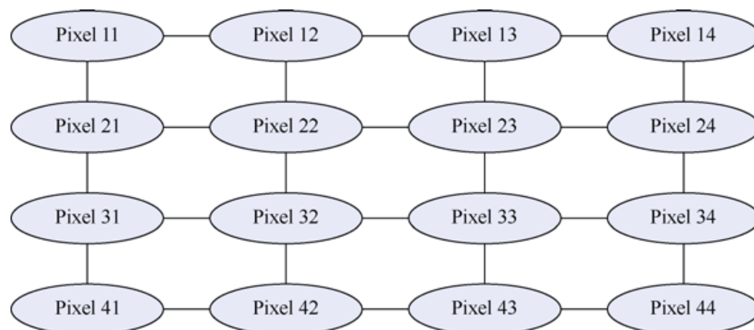


图 10-5　第 5 章中的图像恢复网络重现。这种网格的 VE 与一侧的图像数量成指数关系

分解有许多父变量的 CPD

造成 VE 代价的另一个因素是开始时的因子大小。即使没有增加节点，VE 算法也可能因为有大的因子而长时间运行。如果原始因子来自 CPD，CPD 中提及的变量等于父变量数量加 1（子变量）。这意味着因子的大小与父变量的数量成指数关系。因此，您应该努力减小父变量的数量。

发生这种情况的实际场景之一是 Apply 元素。Figaro 提供的 Apply 元素最多可以有 5 个参数。如果您有一个 5 个参数的函数，就会有一个包含 6 个变量的因子。如果这些参数每个有 10 种可能值，参数值的组合就有 100000 种。这些因子很快就会变成巨型因子。

对 Apply 常常有效的一种技巧是将其分解为多个函数。例如，假定您想将 4 个二项分布相加，可以使用如下代码：

```
val x =
  Apply(Binomial(10, 0.1), Binomial(10, 0.2),
        Binomial(10, 0.3), Binomial(10, 0.4),
        (x1: Int, x2: Int, x3: Int, x4: Int) => x1 + x2 + x3 + x4)
```

每个二项分布有 11 个可能值（0～10），所以参数值的组合有 14641 种，这会造成一个巨大的因子。更好的表现形式是将上述代码分解为 3 个加法，每个加法有 2 个参数：

```
val x12 = Apply(Binomial(10, 0.1), Binomial(10, 0.2),
                (x1: Int, x2: Int) => x1 + x2)
val x34 = Apply(Binomial(10, 0.3), Binomial(10, 0.4),
                (x3: Int, x4: Int) => x3 + x4)
val x = Apply(x12, x34,(x12: Int, x34: Int) => x12 + x34)
```

这段代码包含 3 个函数，每个函数有两个参数。前两个函数的参数组合只有 121 种。x12 和 x34 的取值为 0～20（10+10），所以第 3 个函数参数值的组合为 21× 21 = 441。这样，因子中总共有 683 行，远远优于之前的 14641 行。

实际上，对于加法这种可能应用到任意数量参数的运算，可以很自然地将其分解为一系列 Apply 运算，每个运算有两个参数。幸运的是，您不必自己实现分解。Figaro 已经在 FoldLeft 构造程序中提供了这一功能。FoldLeft 与 Scala 的 foldLeft 类似，适用于一系列元素的迭代运算。例如，在 Scala 中，您可以使用如下代码：

```
val x = List(1, 2, 3, 4)
val y = x.foldLeft(0)((x1: Int, x2: Int) => x1 + x2)
```

其中的 y 为 x 列表项目的总和。类似地，可以使用 com.cra.figaro.library.compound 包中的如下 Figaro 结构：

```
FoldLeft(0, (x1: Int, x2: Int) => x1 + x2)(
  Binomial(10, 0.1), Binomial(10, 0.2),
  Binomial(10, 0.3), Binomial(10, 0.4)
)
```

这将自动地分解为多个函数，每个函数有 2 个参数。Figaro 还提供了 FoldRight 和 Reduce，分别与 Scala 的 foldRight 和 reduce 类似。

利用封装

如果您曾经使用过面向对象编程，就会熟悉如下思路：封装有助于将对程序其余部分不必要的细节隐藏在对象内部。封装对 VE 也有好处。如果对象的接口对模型的其余部分隐藏对象的所有细节，表现这些细节的所有内部元素就在对象内部被消去，不与模型的其余部分相互作用。消去内部元素的结果是接口上的一个因子。这个因子捕捉所需知晓的对象相关细节，这是模型的其余部分所关心的。

有效利用封装的关键是使接口变小。这是标准的 OO 设计指导方针，所以好的 OO

设计和高效的推理是相辅相成的。这方面的例子之一是第 7 章中的计算机系统诊断模型。在这个模型中，您为打印机、网络、软件和用户创建了单独的对象。每个对象使用单一变量与模型的其余部分交互。例如，Printer 对象通过 State 变量与模型的其余部分交互。这意味着，Printer 对象的所有内部结构不管多么复杂，都可以总结为单一变量上的因子。

为了最大限度地利用这一概念，您可以使用模型的层次化分解。例如，Printer 对象可能包含用于 Paper Feed（进纸）和 Toner（墨粉）的嵌套对象。这些嵌套对象将通过一个小型接口与 Printer 通信，后者也使用自己的小型接口与模型的其余部分通信。下面的代码说明如何实现这一点。需要确定的是，如果您想要进行某个元素的查询或者证据提供，它就不能是私有的。在下面的例子中，您希望发布关于墨粉不足指示灯的证据，所以 tonerLowIndicatorOn 元素被放到嵌套的 Toner 私有类之外，它对于 Printer 类是公共的：

```
class Printer {
  val powerButtonOn = Flip(0.95)

  private val heavyUsage = Flip(0.5)

  class Toner {
    private val adequateColorToner =
      If(heavyUsage, Flip(0.8), Flip(0.95))
    private val adequateBlackToner =
      If(heavyUsage, Flip(0.7), Flip(0.9))
    val adequateToner = adequateColorToner || adequateBlackToner
  }

  private val toner = new Toner
  val tonerLowIndicatorOn =
    If(powerButtonOn,
      CPD(toner.adequateToner,
          true -> Flip(0.1),
          false -> Flip(0.99)),
      Constant(false))
val state =
     Apply(powerButtonOn, toner.adequateToner,
           (power: Boolean, toner: Boolean) => {
             if (!power) 'out
             else if (toner) 'good
             else 'poor
           })
}
```

Printer 接口的一部分

封装于 Printer 内部

封装于 Toner 内部

Toner 接口的一部分

封装在 Printer 内部的嵌套对象

放在 Toner 之外，使其可见于模型的其余部分

Printer 接口的一部分

您还可以在不明确使用 OO 设计的情况下得到封装的好处。链结构也提供封装。链

涉及一个父元素和一个子元素，对于父元素的每个值，在对父元素值应用链函数时创建一组元素。只要这些元素不使用链之外的元素，它们就通过链的父元素和子元素封装在链中。如果使用了链之外的元素，使用的元素将成为接口的一部分。所以，只要保持较小的元素数量，就仍然可以从封装中获益。确实，前一小节简短介绍的结构化 VE 明确地使用这种封装。

简化网络

如果您的网络对于 VE 来说过于复杂，但是希望使用精确的 VE 算法，就必须简化网络。这可以通过删除边或者删除节点来完成。您必须判断哪些边和节点与模型的相关性最低，删除能造成最多节省。简化模型的一种替代方案是使用近似推理算法，但是使用简化模型的 VE 的好处是，您将得到与模型相关的准确答案，可以精确地控制模型中包含的内容。

10.3.3 VE 的应用

因为 VE 是完美计算您所感兴趣的概率的准确算法，您可能认为它不适合于具有复杂模型的现实应用。情况并非如此。VE 得到广泛的应用，关键的问题不是模型的规模，而是模型是否有合适的结构——特别是，是否能够消去变量且不在 VE 图中增加太多的边，保持 VE 图中最大团的规模较小，复杂度较低。本小节描述广泛应用 VE 的两个领域。

语音识别

符合 VE 要求的一类模型是第 8 章中介绍过的隐含马尔科夫模型（HMM）。图 10-6 展示了从第 8 章重现的一个 HMM。HMM 是一个动态概率模型，包含一个随时间变化的状态变量和每个时点依赖状态变量的观测值。

HMM 有许多应用，流行的应用之一是语音识别。在语音识别中，您通常有一个由一组随时间变化的音频信号组成的观测序列，希望推断其中说出的单词。

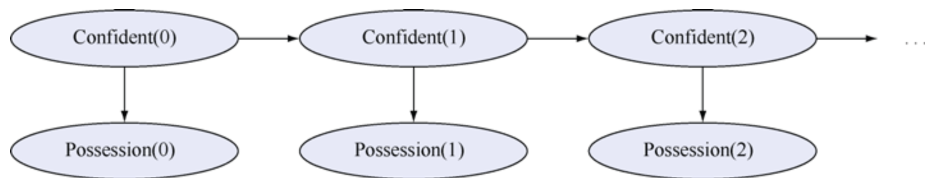


图 10-6 第 8 章中的隐含马尔科夫模型重现

每个单词的模型是一个 HMM，模型中有一系列状态，对应于说话者在说出单词期间发出的声音类型。例如，如果说话者说出“car”这个单词，他可能首先发出“k”的音，然后是“ah”和“r”的音。发出这些音所花费的时间是不确定的。某个音是否存在也是不确定的；例如，有些人不发“car”中“r”的音。这种不确定的发音过程被编码为 HMM 的迁移模型。与此同时，观测模型编码声音信号特征依照一定的概率取决于说

话者所发的音。这样，单词的表达很好地模型化为 HMM。

在做出关于 HMM 的推理时，您通常有一个特定的观测序列（如一定长度的声音信号），希望推理出隐含的状态。例如，您可能有一个长度为 4 的序列，可以将 HMM 展开到 4 个时间步，得到如图 10-7 所示的贝叶斯网络。

关于这个网络，值得注意的是所有节点的父节点都不超过 1 个。这说明当您构造该网络的端正图时，不需要连接任何节点，从而形成图 10-8 所示的图。

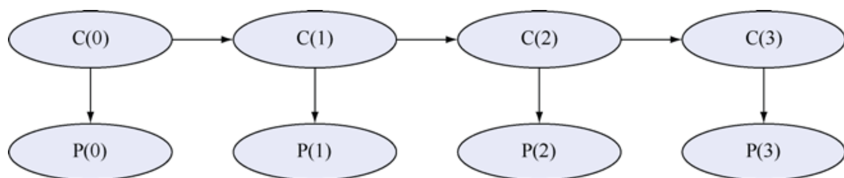


图 10-7　展开为 4 个时间步的 HMM 贝叶斯网络

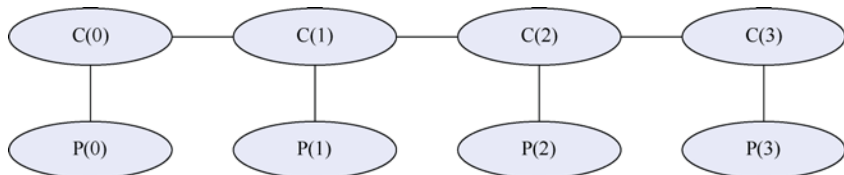


图 10-8　图 10-7 中 HMM 的端正图

下一个需要注意的是，图 10-8 中的图没有环。如您所见，这意味着有一个变量消除顺序不需要在图中增加任何边。例如，假定您对 C(2)的后验概率感兴趣，可以按照如下顺序消除变量：P(0)、C(0)、P(1)、C(1)、P(2)、P(3)、C(3)。按照这个顺序，每当消除一个变量，它就处于图的边缘并且只连接一个其他变量，所以不需要增加边。所以，图 10-8 中的图也是这一变量消除顺序的 VE 诱导图。

仔细观察图 10-8 就会发现，相邻的一对隐含状态相连，每个隐含状态及其对应观测值也是如此。所有这些连接组成了顶点数为 2 的团。没有全部互相连接的 3 个变量，所以最大团顶点数为 2。不管观测序列有多长，将 HMM 展开为多少个时间步，上述情况都成立。因此，HMM 中 VE 的代价与观测序列长度成线性关系。这就是 VE 对于 HMM 是有效算法的原因。

语音识别中用到 HMM 推理的一个变种。首先，您通常对特定隐含状态不感兴趣。其次，您希望知道给定 HMM 下观测序列的概率。例如，假定您不确定说话者说的是“car”还是“jar”，希望确定哪一个单词更有可能。您可以使用贝叶斯推理实现这一目标。回忆第 9 章中的内容，贝叶斯推理中某个事物的后验概率与其先验概率和似然率的乘积成正比。

■　先验概率可能表达“car”比“jar”更常用这一事实，所以在缺乏证据的情况下，

您相信“car”的可能性更大。先验通常从数据中学习而得（人们使用单词“car”和“jar”的频度）。但是在对单词进行分类的时候，先验概率是给定的。

- 与此同时，“car”的似然率是已知单词为“car”的情况下观测到的语音序列的概率，“jar”的情况也类似。这个概率由“car”或“jar”的 HMM 定义。换言之，为了计算似然率，您应该计算 HMM 中观测序列的概率。

实现上述计算的简单方法之一是消去所有变量，观察最后的规格化因子。这个规格化因子等于证据的概率。遗憾的是，Figaro 目前没有访问这个规格化因子的方便接口，但是我希望很快就能增加这个接口。不过，Figaro 提供了其他用于计算证据概率的算法，我们将在第 12 章中详细讨论。特别是，其中有一个基于置信传播的证据概率算法，它的表现本质上和 HMM 上的 VE 相同。

要在语音识别这样的应用中使用 HMM，还必须从数据中学习模型参数。数据通常包含音频序列示例和所说单词的标签。HMM 参数的学习通常也结合 VE 和**期望最大化**（expectation maximization，EM）算法实现。您将在第 12 章中学习如何使用 EM 算法学习模型参数。

自然语言理解

HMM 不是适合 VE 推理的唯一常见结构。在自然语言处理中，往往使用语法分析树（也称解析树）表示句子的组成方式。句子的构造方法被称作**语法分析（也称解析）**。

图 10-9 展示了句子“The cat drank milk”（猫喝牛奶）的语法分析树示例。语法分析的最高级别是符号 Sentence（句子），对应于整个句子。语法分析显示这个顶级符号如何分解为名词短语和动词短语符号。名词短语包含单词“The cat”，动词短语包含单词“drank milk”。这样，语法分析描述了句子的层次结构。

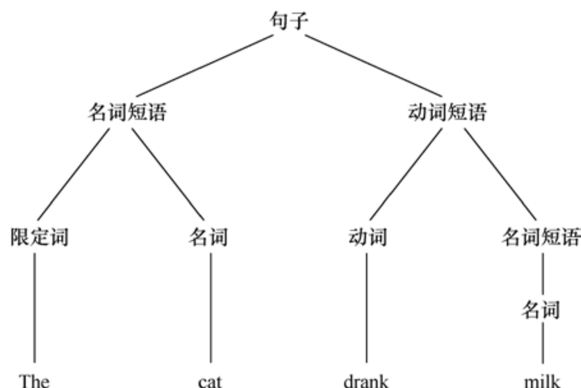


图 10-9 句子“The cat drank milk”的语法分析。这个句子由一个名词短语和一个动词短语组成，动词短语又由一个动词（单词“drank”）和一个名词短语组成

在自然语言理解中，常见的任务之一是确定给定句子的正确解析，这有助于理解句

子。**语法**编码了构造句子的规则。这些规则往往涉及选择。例如，动词短语可以由一个动词（如句子“The cat sat.”）或者由一个动词跟上一个名词短语（“The cat drank milk.”）组成。**概率语法**用概率表示这些选择，人们已经发现它在生成正确解析上很有效。例如，动词短语仅由动词组成的概率可能为 40%，而由一个动词短语加上一个名词短语的概率为 60%。

概率上下文无关语法（probabilistic context-free grammar，PCFG）是一种特别简单的概率语法，因为简洁和易于推理而大受欢迎。在此不对 PCFG 的定义详细解释，但是 PCFG 的主要特点是语法分析中的不同时点做出的决策相互独立。这是使 VE 可以在 PCFG 上有效工作的必备属性。

PCFG 的思路是，对于句子的每个非空子串，都有一个变量表示子串相关的符号。在我们的例子中，这些子串是 The、cat、 drank、 milk、 The cat、 cat drank、 drank milk、The cat drank、 cat drank milk 和 The cat drank milk。根据 PCFG 的规则，因为“The cat drank milk”可以由“The cat”和“drank milk”组成，“The cat drank milk”的符号可能影响“The cat”和“drank milk”的符号。这一通用过程定义了所有变量上的贝叶斯网络。

在这个网络上，您可以从底向上消除变量，确定一个句子的解析。例如，您可以从对应于各个单词的变量开始。在消去所有变量之后，可以消去对应于长度为 2 的子串的变量。根据 PCFG 的独立性假设，这些变量也都可以单独消去。按照这一方式继续向上；在任何时候，消除长度低于 n 的所有子串对应的变量之后，可以独立地消去长度为 n 的子串对应的变量。

这样做的结果是，基于 PCFG 的 VE 代价与句子长度的立方成正比。对于大部分应用中遇到的典型句子，这是可行的，所以它成为了广为使用的算法。往往，您对符号上的概率分布并不感兴趣，而是希望推导**最可能的解析**。这可以归入最可能解释查询的类别。在第 12 章中，您将学习这些查询的有关知识，包含用于计算 PCFG 最可能解释的 VE 算法。

10.4　置信传播

当您能够完成 VE 的运行时，它产生准确的结果。遗憾的是，有时候您可能不愿意对模型进行足够的简化，以运行 VE。您已经知道，VE 的复杂度和所使用变量消除顺序的诱导图中最大团的顶点数成指数关系。如果必须增加过多的边而使诱导图过于密集怎么办？幸运的是，被称作置信传播（belief propagation，BP）的近似算法能够很好地处理这种情况。本书不介绍 BP 的完整细节，但是我将介绍足以帮助您理解其工作方式和使用场合的基本原理。

10.4.1 BP 基本原理

BP 使用端正图（初始 VE 图）运行，不增加边。只要端正图中最大的团不太大，BP 就是高效的。由于不需要增加边，BP 可以避免创建很大的团，这使它能够实现比 VE 更快的推理速度，但是需要付出一定的代价。增加这些边对于正确的推理是必要的，所以不增加边将导致误差。不过，即使不增加这些边，推理也可以近似于正确。

在 10.2.1 小节中您已经学到，如果端正图是三角化的，就有某些变量消除顺序在产生诱导图时不需要增加边。因此，如果在三角化图上运行 BP，它产生的是正确的答案。实际上，BP 和 VE 在三角化图中的复杂度相同。但是如果在非三角化图上运行 BP，它就是一个近似算法，可能远比 VE 高效。非三角化图上的 BP 称作**多环（Loopy）** BP 算法。

BP 是一种**消息传递算法**，理解它的最好方式是考虑在三角化图上的运行，在这种情况下它是精确算法，和 VE 一样解乘积之和表达式。但是它并不通过使用代数规则操纵表达式求解，而是通过在网络节点之间传递消息解题。这些消息基于因子运算。这里，我不打算介绍消息的计算方法，但是这些消息模拟 VE 执行的因子乘积和求和运算。

不过，BP 和 VE 之间存在重大差异。**BP 同时为网络中所有变量上的查询执行这些计算**。在运行 BP 之后，可以得到给定证据情况下任何变量的后验概率。在三角化网络中，只要您组织消息以正确的方式传递，BP 可以在网络上的两次传递中实现上述目标。（遗憾的是，您将会看到，Figaro 无法组织消息实现该目标，所以需要两次以上的传递，但是这一属性在 BP 算法的原理上是成立的）这是 BP 相对 VE 的重要优势，即使在三角化图中，如果您感兴趣的是查询多于一个变量，这一优势就存在。

在非三角化网络中，Loopy BP 的工作方式和 VE 相同，但是它不会在两次传递（如果消息组织得当）后终止，而是在您所指定的时间长度内重复执行。消息在网络上传递，如果端正图有多个环，消息还将在环中传递。您可以重复任意次运行 BP，得到给定次数重复之后的最佳答案。理想情况下，您重复运行的次数越多，得到的答案越接近正确答案。

10.4.2 Loopy BP 的属性

Loopy BP 的有效性有些令人吃惊，其原因和统计物理学有关。简言之，Loopy BP 计算网络的“贝特自由能”；在此我不对这个概念进行详细的介绍。贝特自由能可用于近似计算网络中变量的后验分布，但是不能得到完全相同的结果。

随着 Loopy BP 的迭代次数越来越多，在许多情况下，它应该收敛于贝特自由能。遗憾的是，贝特自由能与后验概率的近似程度得不到保证。“理解 Loopy BP 的精度”补充材料提出了两个几乎完全相同的示例程序。其中一个程序中 Loopy BP 得到正确的答案，在另一个程序中则差得很远。

理解 Loopy BP 的精度

考虑下面两个程序。

好的 Loopy BP 示例

```
val e1 = Flip(0.5)
val e21: Element[Boolean] = Apply(e1, (b: Boolean) => b)
val e31: Element[Boolean] = Apply(e21, (b: Boolean) => b)
val e22: Element[Boolean] = Apply(e1, (b: Boolean) => b)
val e32: Element[Boolean] = Apply(e22, (b: Boolean) => b)
val e4: Element[Boolean] = Dist(0.5 -> e31, 0.5 -> e32)
println(BeliefPropagation.probability(e4, true))
println(VariableElimination.probability(e4, true))
```

这在 e31 和 e32 之间随机选择

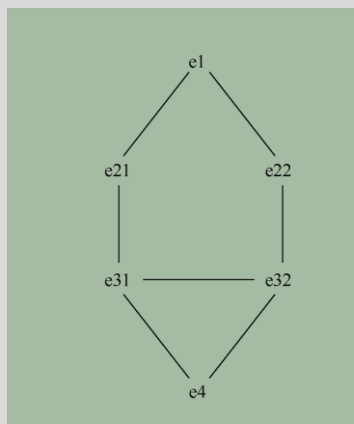
不好的 Loopy BP 示例

```
val e1 = Flip(0.5)
val e21: Element[Boolean] = Apply(e1, (b: Boolean) => b)
val e31: Element[Boolean] = Apply(e21, (b: Boolean) => b)
val e22: Element[Boolean] = Apply(e1, (b: Boolean) => b)
val e32: Element[Boolean] = Apply(e22, (b: Boolean) => b)
val e4: Element[Boolean] = e31 === e32
println(BeliefPropagation.probability(e4, true))
println(VariableElimination.probability(e4, true))
```

当且仅当 e31 和 e32 的值相等，e4 为真

这两个程序首先打印 BP 产生的“e4 为真”概率，然后打印 VE 产生的准确答案。近似计算做得越好，与答案的距离就越小。如果运行第一个程序，两个算法都将产生答案 0.5，也就是说这一近似计算是完美的。对于第二个程序，BP 的结果为 0.5，VE 的结果为 1.0，近似计算的结果很糟糕。

发生了什么事？两个程序的结构完全相同。这些程序的端正图可以在下图中看到。实际上，两个程序的唯一差异是 e4 的定义。



Loopy BP 示例的端正图，e31 和 e32 之间存在一条边，因为它们都是 e4 的父变量。该网络包含一个没有交叉边的环 e1–e21–e31–e32–e22，所以它是非三角化图

仔细观察这些定义可以发现，秘密就在 e32 和 e31 决定 e4 时相互作用的程度。在第一个程序中，e4 在 e31 和 e32 之间随机选择。e31 和 e32 在决定 e4 时没有相互作用；如果 e31 因为被选中而影响 e4，则 e32 没有影响，反之亦然。在第二个程序中，当且仅当 e31 和 e32 值相等时 e4 为真。e31 和 e32 在决定 e4 时有极端的相互作用；只要看看 e4 的定义就知道，除非知道 e32 的值，否则知道 e31 的值无法告诉我们关于 e4 值的任何信息。但是实际上我们知道 e31 和 e32 肯定相等，因为它们都（间接）等于 e1。所以 e4 的值肯定为真，这就是 VE 产生答案 1.0 的原因。

这就为我们指出了关键要点。程序 2 中得到正确答案需要非局部推理：跟踪 e31 和 e32 对 e1 的共同依赖性。BP 无法进行非局部推理，它在长距离效应很小的情况下发挥作用。在许多问题上都是如此，变量在长的路径上效果将会衰减。这就是 BP 往往在典型模型上表现很好的原因，实际上，程序 2 是人为的例子。我必须使 e21 和 e1 完全相同，e31 与 e21 完全相同才能证明这一点。典型的模型可能不是这样定义的。

关于 Loopy BP 的性能还需要说明一点：它甚至不能保证收敛。有时候，消息在环中传递时，它们在一次迭代和下一次迭代之间摇摆不定。幸运的是，这种情况不太常见。更重要的问题是 Loopy BP 近似值有时收敛于和正确的后验分布相去甚远的位置。

结论：Loopy BP 是一种实用算法，在许多情况下可以快速运行，生成相当精确的近似值。但是总体来说，它对结果没有任何保证，不能保证收敛，也不能保证收敛得到的近似值的精度。因此，常见的策略如下。

1．尝试 VE。如果 VE 能在合理的时间内完成，您就能得到准确的答案，这是最好的结果。

2．如果 VE 不能在足够短的时间内结束，尝试 BP。检查它给出的答案是否合理。您很有可能对答案有某种直觉，或者可以使用试验数据检查答案。

3．如果 BP 也太慢，或者对其产生的答案不满意，尝试下一章讨论的抽样算法。

10.5 BP 的使用

和 VE 一样，我将首先介绍在 Figaro 中使用 BP 的各种选择，然后提供有效使用 BP 的一些通用原则，最后介绍一些实际应用。

10.5.1 Figaro 特有的 BP 考虑因素

运行 BP 的关键问题是采用多少次迭代。运行 BP 时，您有 3 种选择。

■ 指定迭代次数。可以将迭代次数作为算法创建的一个参数，例如：

```
val algorithm = BeliefPropagation(100, element)
algorithm.start()
println(algorithm.probability(element, true))
```

- 使用任意时间算法。在这种情况下，您不指定迭代次数，而是运行任意的时长。为了唤醒您对任意时间算法的记忆，可以运行如下程序：

```
val algorithm = BeliefPropagation(element)
algorithm.start()
Thread.sleep(1000)
println(algorithm.probability(element, 0))
Thread.sleep(1000)
println(algorithm.probability(element, 0))
algorithm.kill()
```

第一次答案估算

等待 1 秒

第二次答案估算（结果可能更好）

完成时杀死任意时间算法以释放线程很重要

- 使用单行快捷方式如 BeliefPropagation.probability(element, 0)。这种快捷方式使用默认迭代次数；如果想要指定迭代次数，就必须使用长形式。

应该采用多少次迭代？Figaro 使用 BP 的异步形式，在这种形式中，消息在网络中传递的顺序不受控制。一般来说，如果想保证信息从一个节点到达另一个节点，迭代数量应该至少等于节点之间的距离。在 10.4.1 小节中我曾经说过，在一个非多环网络中，BP 可以在网络上的两次传递中得到正确答案。对于 Figaro，这意味着迭代次数应该至少两倍于网络直径加 1。（网络直径是两个节点间的最大距离）也就是说，Figaro 的 BP 在直径较小的网络上更高效。

对于 Loopy BP，情况也类似。如果您使用的迭代数量等于网络直径，就等同于绕网络一周。所以如果自己选择迭代数量，应该用直径乘以想要的圈数。迭代数量不一定是准确的；只需要选择确保您在网络中来回多次的迭代次数。常用的经验法则是绕网络 10 圈。如果不想操心这个参数，可以使用该算法的任意时间版本。

10.5.2 设计模型以支持高效的 BP

BP 的复杂度取决于端正图中最大团的顶点数，而不是 VE 的诱导图。而且，您知道 BP 在端正图是三角化图时是精确算法。VE 中的许多考虑因素也适用于 BP，但是也有一些 BP 特有的考虑因素。

避免过多的环

虽然环对 BP 复杂度的影响不像 VE 中那样大，但是确实会给 BP 带来误差，您已经了解，这种误差可能难以估计。环越多，误差的可能性越大。因此，避免过多的环仍然是一个好主意，但是不像 VE 中那么重要。

合并元素

避免环的方法之一是合并节点，人工消除环。我们再来看看上面的补充材料中不好

的程序：

```
val e1 = Flip(0.5)
val e21: Element[Boolean] = Apply(e1, (b: Boolean) => b)
val e31: Element[Boolean] = Apply(e21, (b: Boolean) => b)
val e22: Element[Boolean] = Apply(e1, (b: Boolean) => b)
val e32: Element[Boolean] = Apply(e22, (b: Boolean) => b)
val e4: Element[Boolean] = e31 === e32
```

您可以将 e21 和 e22 元素合并为元素 e2，e31 和 e32 合并为元素 e3，如图 10-10 所示。因为 e21 和 e22 都是布尔元素，e2 是 Element[(Boolean, Boolean)]，其值是 e21 和 e22 值的配对。因此，e2 的可能值数量为 2 × 2 = 4；e3 与此类似。

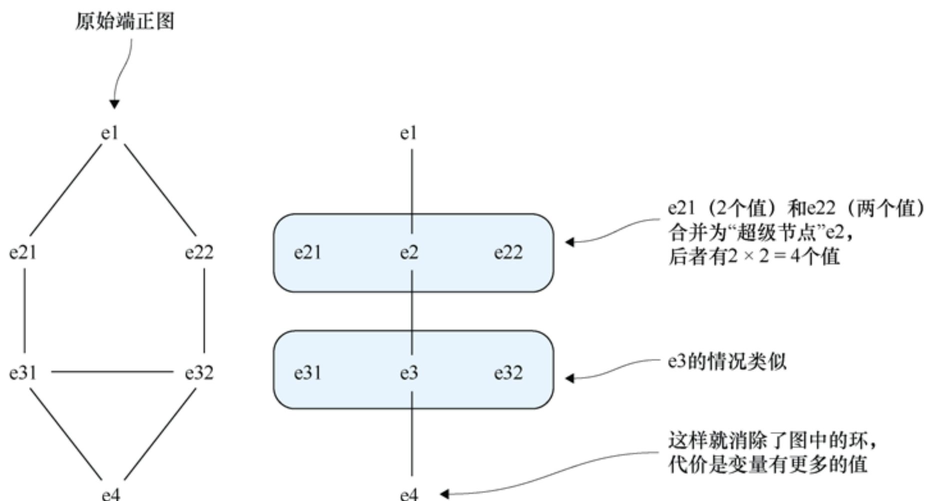


图 10-10　合并节点以避免环。左边是原始的端正图，用 BP 会生成不精确的结果。右边是合并了节点的图，没有环

您可以编写如下程序：

```
val e1 = Flip(0.5)
val e2: Element[(Boolean, Boolean)] = Apply(e1, (b: Boolean) => (b, b))
val e3: Element[(Boolean, Boolean)] =
  Apply(e2, (bb: (Boolean, Boolean)) => (bb._1, bb._2))
val e4: Element[Boolean] =
  Apply(e3, (bb: (Boolean, Boolean)) => bb._1 == bb._2)
```

创建一个配对元素，配对的每个成分等于 e1 的值，就像原始程序中的 e21 和 e22

创建一个配对元素，其中第一个成分等于 e2 第一个成分的值，第二个成分也类似

很明显，这不像原来的程序那么自然、易读。但是这段程序的主要好处是 BP 可以得到正确的答案。

分解有许多父变量的 CPD

因为 BP 在端正图上工作，端正图对每个 CPD 有一个团，和 VE 一样，有许多父变量的 CPD 对 BP 也是很大的问题。特别是，BP 的复杂度与节点最大父变量数成指数关系。因此，您应该尝试用 10.3.2 小节中描述的技术分解 CPD。

使用衰减 CPD

如果有长距离效应，Loopy BP 的误差会变得更大，在这种情况下，节点的值通过环对网络中距离很远的另一个节点的值产生很大影响。当路径上的变量之间是确定性的，这种长距离效应最强。您在 10.4.2 小节的补充材料中已经看到了一个例子，该例中的中介变量等于第一个变量。用具有某种随机性的 CPD 缓和这种确定性，可以减小长距离效应。

衰减 CPD 不仅能够得到更好的 BP，还能得到更精确的模型。例如，考虑第 5 章中的打印机模型。在这个模型中我们说明，如果打印机电源按钮开启，墨粉水平高，纸张流动顺畅，则打印机处于好的状态。这是一种确定性的关系。但是情况是否始终如此？实际上不是，其他电气或者机械故障可能导致打印机无法正常工作。我们往往用确定性的关系简化模型，但是增加噪声可能更加准确。而且，因为这能够得出更好的 BP，所以通常是一个好主意。在下一章中，您将发现增加少数噪声也能改进采样算法的精度。

简化网络

当其他选择都无效时，简化网络可能是一个好主意，这和 VE 的情况相同。简化网络时，您应该有两个目标，最重要的目标是，减少 CPD 中父变量的数量，使推理更快。其次，尽可能消除长距离效应——例如，通过删除没有交叉边的环进行。

10.5.3　BP 的应用

模型必须有明确的结构，VE 算法才适用。您已经在 HMM 和 PCFG 的例子中看到了这种结构。BP 无此限制。因此，BP 的应用很广泛。每当您有一个包含离散变量的模型，BP 就是一个好的候选技术。即使您有连续变量，只要愿意将它们限制在一组特定的可能值，也可以使用 BP。下面是 BP 的一些常见应用。

- **图像分析**——在第 5 章中您已经看到，可以使用马尔科夫网络建立二维像素数组的模型。图 10-5 展示了 4×4 图像的马尔科夫网络示例。在那一节中我曾经说过，这类网络对 VE 来说很困难，因为推理的代价与图像的大小成指数关系。相反，BP 很适合于这类模型。推理的代价与图像的大小仅成线性关系。
- **医学诊断**——在典型的医学诊断问题中，患者报告一组症状，医生需要找出这些症状的根源。图 10-11 展示了一个用于医学诊断的贝叶斯网络示例。患者有

一组可能的疾病并报告了一组症状。每种症状可能是多种疾病造成的。这些贝叶斯网络通常有很多环。一般来说，随着疾病和症状数量的增长，图中的团变得很大，VE 不适合于这种情况。BP 已经成功地用在这类网络上。

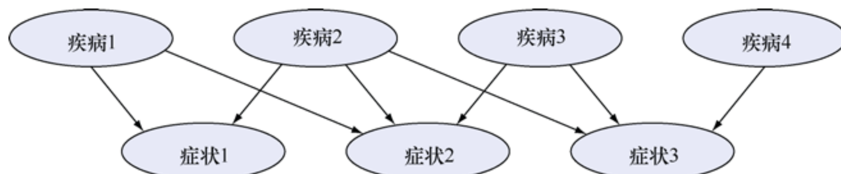


图 10-11　医疗诊断贝叶斯网络

- **建筑、车辆或者设备健康监控**——复杂系统健康状况的监控是概率推理的重要应用。例如，假定您有一个数据中心，希望监控各个组件的温度和电源使用情况。您可能有表示每个组件温度和电源使用情况的变量。附近组件的温度也是相关的，因为热量可能在这些组件之间传递。相互连接组件的电源使用情况变量也是相关的。此外，组件的电源使用影响其温度。这些因素创建了一个大的多环概率模型。在这个例子中，温度和电源使用是连续变量，必须离散化为一组值。不过，BP 在这类应用中很有效。

10.6　小结

- 因子是组织概率推理期间计算的数据结构。
- 因子有一组变量、与每个变量值组合对应的行和对应每行的一个数值。
- 因子支持乘积和加总运算，这些运算通过对应条目相乘和加总定义。
- 概率模型查询的答案可以定义为因子的乘积之和表达式。
- 变量消除法是一种精确算法，智能地操纵乘积之和表达式，而不创建完整的联合分布。
- 变量消除法的复杂度取决于变量消除顺序；对于给定的变量消除顺序，复杂度与诱导图中最大团的顶点数量成指数关系。
- 置信传播算法使用因子运算传递消息。在三角化网络中，它是一种精确算法。在有环网络中，这是一种实用的近似算法，但是不能提供任何保证。
- 多环置信传播在网络中的环的长距离效应较小的情况下更为精确。

10.7　练习

本章的前 5 个练习基于图 10-12 所示的贝叶斯网络，描述了婴儿生命期的一部分。图

中展示了 4 个变量的 CPD。在 www.manning.com/books/practical-probabilistic-programming 可以找到部分练习的答案。

1．将每个 CPD 写成一个因子。

2．运用链式法则，编写所有变量联合概率分布的表达式。将所有因子相乘计算联合概率。

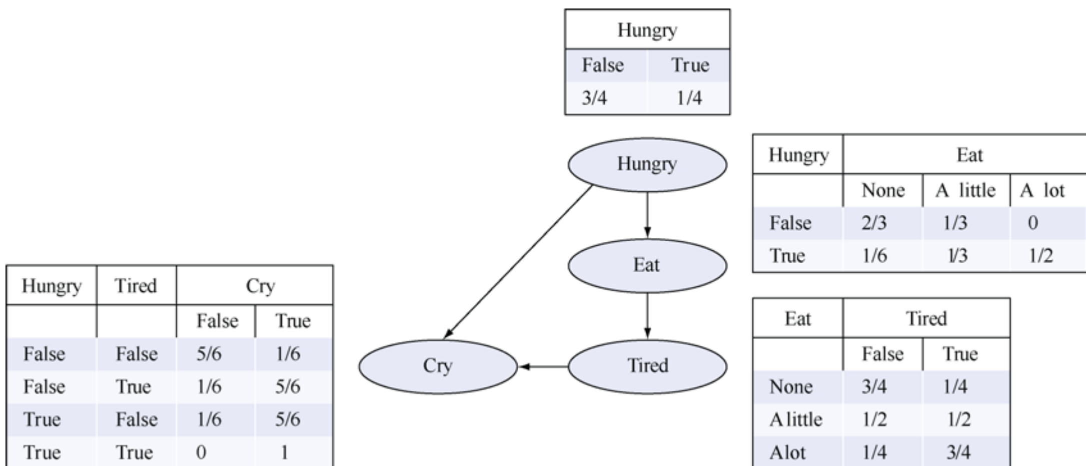
3．用全概率公式编写 P(Cry)的表达式。加总联合分布中的 Hungry、Eat 和 Tired，计算 P(Cry)。

4．编写查询 P(Eat | Cry = True)的表达式。从联合分布入手，按照如下步骤计算查询的答案：

a）将 Cry 不为 True 的行设置为 0。

b）加总变量 Hungry 和 Tired。

c）规格化结果。



Hungry	
False	True
3/4	1/4

Hungry	Eat		
	None	A little	A lot
False	2/3	1/3	0
True	1/6	1/3	1/2

Eat	Tired	
	False	True
None	3/4	1/4
A little	1/2	1/2
A lot	1/4	3/4

Hungry	Tired	Cry	
		False	True
False	False	5/6	1/6
False	True	1/6	5/6
True	False	1/6	5/6
True	True	0	1

图 10-12　婴儿生命期贝叶斯网络

5．使用变量消除法，考虑计算 P(Cry)的过程。

a）绘制贝叶斯网络的端正图。

b）按照变量消除顺序 Eat、Hungry、Tired 绘制诱导图。最大团的顶点数量是多少？

c）按照变量消除顺序 Hungry、Tired、 Eat 绘制诱导图。最大团的顶点数量是多少？

6．对于图 10-5 中的 4×4 像素网络，尝试不同的变量消除顺序。说服自己相信无法避免至少为 4 的团顶点数量。

7．用 Figaro 表现 HMM，例如图 10-6 中的模型。编写一个函数，以观测序列为参数，在这个观测序列上展开 HMM，使用变量消除法计算最终隐含状态的概率分布。计量不同长度的观测序列的计算时间。以观测序列长度的函数表示，计算时间遵循何

种趋势？

8．在 Figaro 中创建图 10-5 中的像素网络的通用表现形式，每边的像素数量可变。对于不同像素数量，进行如下试验。

a）随机观察 1/3 像素，不包括左上角。

b）计量变量消除法计算左上角有像素概率所花费的时间。您将发现，在超出一个相对小的像素数之后，变量消除法将花费很长时间，因此停止该过程。变量消除法的计算时间遵循什么趋势？

c）计量置信传播计算左上角有像素概率所花费的时间。置信传播遵循什么趋势？

d）如果变量消除法和置信传播都终止，计量两种算法产生的答案之间的差异。因为变量消除法得出精确的答案，这一数量就是置信传播的误差。

9．考虑图 10-11 中的医学诊断网络，将疾病放在一行，症状放在另一行。

a）编写一个函数生成这类随机网络。该函数应该有 3 个参数：疾病数量、症状数量，任意给定的疾病和症状之间有一条边的概率。

b）和练习 8 一样进行试验，观察参数值变化时变量消除法和置信传播法的表现。变量消除法的计算时间有何趋势？置信传播的计算时间有何趋势？当两个算法终止时，置信传播的误差如何？

第 11 章　抽样算法

本章介绍如下内容：

- 抽样算法的基本原理
- 重要性抽样算法
- 马尔科夫链蒙特卡洛（MCMC）算法
- MCMC 的 Metropolis-Hastings 变种

本章继续前一章的主题，介绍用于概率编程推理的一些主要算法。第 10 章的焦点是变量消除法和置信传播等因子分解算法，本章则研究抽样算法，这类算法生成从程序定义的概率分布中抽取的变量可能状态以回答查询。本章要特别介绍两种实用的算法：重要性抽样和马尔科夫链蒙特卡洛算法（MCMC）。

完成本章的学习之后，您将对概率编程系统（如 Figaro）使用的推理算法有很好的理解。这种理解将帮助您设计模型、控制推理，以得到更好的结果。特别是，MCMC 需要额外的努力才能有很好的表现，本章介绍了这方面的几种技术。第 12 章在这些知识基础上，说明如何使用类似的算法回答概率程序上的其他查询。

本章和前面介绍因子分解算法的章节大致上是相互独立的。尽管 我将与因子分解算法进行一些对比，但是理解抽样算法并不要求理解因子分解算法的工作原理。而且，抽象算法较少使用第 9 章中介绍的推理原则。当然，您应该对编写 Figaro 程序有一定的理解。最后，MCMC 算法在第 8 章开始时介绍的马尔科夫链基础上构建，所以如果想要理解 MCMC，应该复习那些材料。

11.1　抽样的原理

抽样算法是概率推理中因子分解算法的一种替代算法。抽样的基本原理很简单：使用可能世界的一组示例（**样本**），而不是以一组数值的形式表现可能世界上的概率分布。

抽样的基本思路如图 11-1 所示。在这个例子中，一个变量的可能取值为 small（小）、medium（中）和 large（大）。可能世界由这个变量的可能值组成，第一行显示可能世界上的真正概率分布。这种真实分布可能是查询的答案，但通常是未知的。您并不直接计算分布，而是生成一组样本。每个样本是一个可能世界，生成特定可能世界的概率应该等于该可能世界的概率。图中，我们生成了：

- 该变量设置为 small 的 2 个样本。
- 该变量设置为 medium 的 5 个样本。
- 该变量设置为 large 的 3 个样本。

然后，您可以通过等于该可能世界的样本所占比例估算可能世界的概率：

- 您希望计算一个真实分布。
- 生成来自这个真实分布的样本。
- 使用这些样本估算分布。

真实概率	0.23	0.46	0.31
样本	··	·····	···
估算的概率	0.2	0.5	0.3
值	small	medium	large

图 11-1　抽样原理。您希望计算一个未知的真实概率分布。为了估算这个分布，生成一组样本。生成特定值的概率应该等于该值的真实概率。在得到一组样本之后，可以通过观察有多少样本取得某个值，估算该值的概率

前面的例子中有一个仅有 3 种可能取值的离散变量。抽样也可用于具有无穷多个可能取值的变量，如连续变量。实际上，抽样可能在这类变量上最有用。您可能还记得，连续变量使用概率密度函数。图 11-2 展示了两个 0 和 1 之间的连续变量；每个可能世界包含每个变量的一个值。可能世界的空间被分为不同概率密度的区域。区域的阴影颜色越深，密度越大。这个区域被一组样本覆盖。样本在区域中的密度粗略地反映了该区域中的概率密度。某个区域中样本的数量是该区域概率的一个估算。

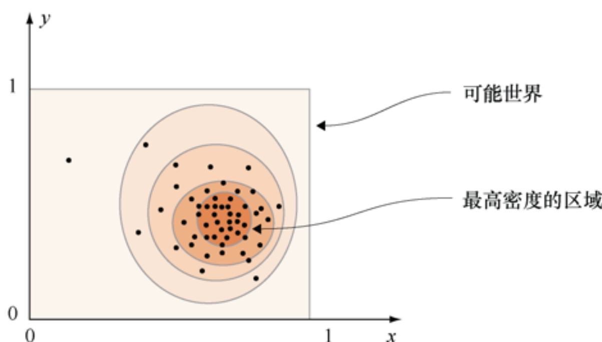


图 11-2　用一组样本覆盖可能世界空间。深色阴影区域是概率密度加高的区域。高密度区域的样本密度高于低密度区域

抽样算法有两个主要的类别。第一类算法较为简单，直接运行概率程序从预想的分布中生成一个样本。这类算法称作**前向抽样**，下面将做介绍。许多概率编程系统（包括 Figaro）中实现的一种实用前向抽样算法是重要性抽样，我们将在 11.2 小节中介绍。

第二类抽样算法称作**马尔科夫链蒙特卡洛（MCMC）算法**。MCMC 的主要思路是不直接从分布中抽样，而是定义一个最终收敛到真实分布的抽样过程。11.3 小节中将详细解释这一过程。

11.1.1　前向抽样

在前向抽样中，一次生成概率程序中的一个变量值。在为所有变量生成一个值之后，您就有了一个由单个样本组成的可能世界。每当生成一个变量值，就使用该变量的定义：使用其函数形式和数值参数依赖于其父变量的方式。因此，变量的定义决定了该变量上的一个概率分布。您根据这个分布为变量选择一个值。前向抽样按照拓扑顺序推进：在为某个变量生成值之前，总是先生成该变量父变量的值，所以您总是明确所使用的分布。

下面是前向抽样的伪代码。这些代码生成一个由所有变量值组成的样本。

1．假设 O 是变量的拓扑顺序。

2．对于 O 中的每个变量 V：

a）设 Par 为 V 的父变量。

b）设 x_{Par} 是之前为 Par 生成的值。

c）从 $P(V \mid Par = x_{Par})$中提取 x_V。

3．返回 **x**（表示每个变量 V 值 x_V 的一个向量）。

前向抽样过程如图 11-3 所示。待抽样的概率程序显示在左侧，中部是一棵展示程序中变量所有可能值生成过程的树。粗箭头表示正在运行的特定抽样。首先，为变量

rembrandt1（表示两幅画中的第一幅是不是伦勃朗的作品）生成值。因为该变量的定义为 Flip(0.4)，生成 true 值的概率为 0.4，false 值的概率为 0.6。在图中所示的采样路径中，生成 true 值。接下来，为 rembrandt2 生成值。这个变量不依赖于任何其他变量，也由一个 Flip 定义，所以过程类似。在这一次运行中，rembrandt2 生成的值是 false。最后，生成代表两幅画作是否同一画家作品的 samePainter 值。（只考虑两位画家，所以如果两幅画作都不是伦勃朗的作品，则它们都出自其他画家之手）变量 samePainter 依赖于 rembrandt1 和 rembrandt2，这两个变量的值此时已经生成。因为 rembrandt1 为 true，rembrandt2 为 false，所以 samePainter 自动为 false。

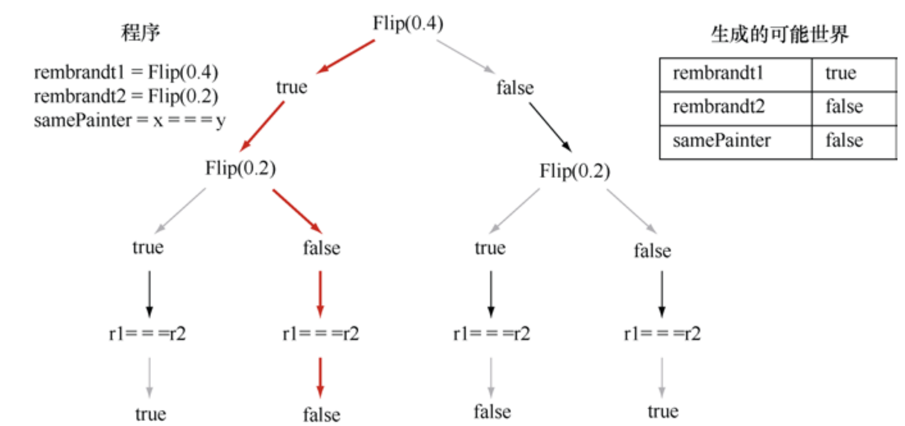


图 11-3　前向抽样过程。图中的树展示了为给定程序生成值的所有可能途径，以及通过树生成某个特定值的过程。右侧是从这次运行得到的可能世界（在树中，rembrandt1 和 rembrandt2 缩写为 r1 和 r2）

在前向抽样中，运行上述过程许多次，每次都生成不同的可能世界（或者样本）。表 11-1 展示了这个程序的 4 个样本。注意，同一个样本可能生成多次；在这个例子中，第二个和第四个样本相同。可能世界的概率可以估算为等于该可能世界的样本比例。例如，rembrandt1 为 false、 rembrandt2 为 false 且 samePainter 为 true 这一可能世界的概率估算为 1/2，因为 4 个样本中有 2 个等于该可能世界。

表 11-1　图 11-3 的程序生成的 4 个样本

元　　素	第 1 个样本	第 2 个样本	第 3 个样本	第 4 个样本
rembrandt1	true	false	true	false
rembrandt2	false	false	true	false
samePainter	false	true	true	true

估算的样本分布也可以用于回答查询。例如，如果您想知道两幅画作是同一作家作品的概率，该怎么做？您可以使用 4 个样本中有 3 个的 samePainter 值为 true 这一事实，估算出查询的答案为 3/4。

为什么使用样本?

强调一组样本只是可能世界上概率分布的一种可能表现形式，是很重要的。前向抽样是一个随机过程，所以它每次都可能生成不同的结果。从前向抽样的不同次运行中得到的估算分布通常也不同。而且，它们通常不会正好等于概率程序定义的分布。实际上，观察上述的例子，Flip(0.4)为 false 的概率是 0.6，Flip(0.2)为 false 的概率为 0.8。两者皆为 false 的概率是 0.6 × 0.8 = 0.48。在这个例子中，Flip(0.4) === Flip(0.2)始终为 true。所以，可能世界等于表 11-1 中第 2 和第 4 个样本的正确概率是 0.48 而不是 0.5。在这个特殊的例子中，估算值很接近，但是通常不能保证答案能够如此接近，特别是在您生成的样本数量不大的情况下。

那么，为什么抽样是个好主意呢？这个例子只有 3 个变量，您可以使用简单的乘法和加法进行所有必要的计算。但是，许多程序有很多变量，这些变量可能采用有大量甚至无穷多个可能值的丰富数据类型。在那些程序中，可能无法创建所有必要的因子以运行分解推理计算法，更不要说执行所有必要的乘法和加法了。另一方面，使用抽样，您只需要考虑相对少的可能性，获得所需概率分布的一个估算值。

下面是体现上述要点的一个示例程序：

```
val x = Apply(Normal(0.4, 0.3), (d: Double) => d.max(0).min(1))
val y = Flip(x)
```

在此，x 代表一个 0 和 1 之间的正态分布变量。变量 y 定义为复合 Flip，与 Chain(x, (d: Double) => Flip(d))等价。图 11-3 展示了完整的可能抽样运行树。但是，在实践中，这棵完整的树从未创建——只生成对应于各个值的子树。在这个例子中，完整的树是无限的，所以永远无法计算精确的概率分布。抽样仅计算树中有限的一部分，并用结果估算分布。

图 11-4 展示了抽样这一程序生成的树。在每次运行中，为 Apply 生成一个实数值，这作为参数传递给 Chain，每次都生成不同的 Flip 元素。如果生成 4 个样本，只会创建 4 个特定的 Flip 元素；其余的无穷多个元素甚至从未存在过。

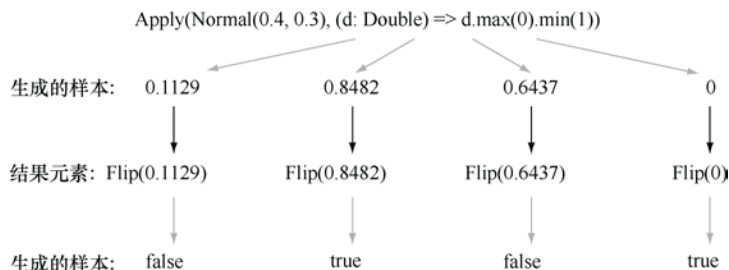


图 11-4 样本生成的部分树

如果抽样的分布只是真实分布的一个估算，对抽样的结果能说什么？这里有两个要点，您可能直观地感觉到它们。

- **平均来说，样本分布将等于真实分布**。任何特定的样本分布与真实分布存在差异，但是如果取所有可能分布的平均值，将等于真实分布。从直觉上说，这是因为每个样本都是从真实分布中提取的。这一属性用技术术语描述为：样本过程**无偏性**。
- **使用的样本越多，样本分布越接近于真实分布**。换句话说，如果您使用样本回答查询，答案的预期误差随着样本的增加而减小。这仅是预期误差；没有任何保证。如果您的运气不好，更多的样本可能造成误差的增加，但是平均起来，误差将会减小。这种属性的术语说法是：**抽样过程的方差随着样本增多而减小**。

图 11-5 展示了抽样算法的典型表现。该图显示了样本数量随时间推移而增大时，抽样算法估算的查询概率。可以看到，一开始估算值快速地向真实概率移动，在长时间运行之后收敛于真实概率，但是有时估算会出现发散，收敛的速度也可能很慢。

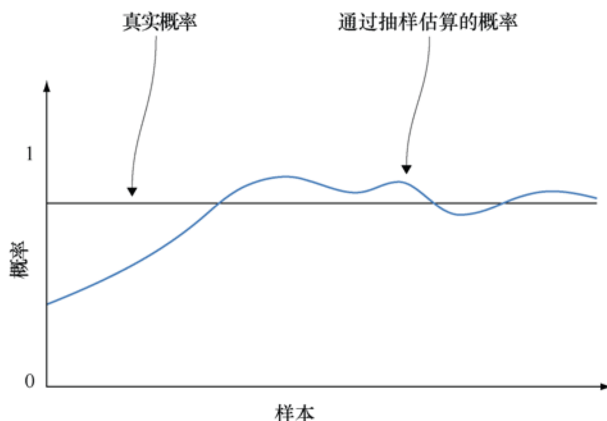


图 11-5 抽样过程在一段时间内的典型表现。抽样估算的概率倾向于收敛到真实概率，但是有时候也会发散

这两个属性是抽样算法成为实用的概率推理近似算法的关键原因。根据这些属性，**如果您使用一个抽样过程，将得到以真实分布为中心的近似分布，可以预期，使用的样本越多，近似分布就越接近于真实分布。如果使用足够多的样本，与真实预期的差距可以达到您的预期**。

警告：在这个问题上没有什么致胜法宝。推理是一个困难的问题，抽样也不总是好的解决方案。有时候，您不得不使用很大的样本，以得到和真实分布足够接近的结果。

在生成一组样本之后，很容易用它们回答查询。例如，为了估算变量取得特定值的概率，计算样本中变量取该值的比例。要估算变量的最可能值，可以观察样本中最常见的值。所以，可以总结抽样算法如此吸引人的原因：**如果使用足够的样本，可以接近于真实分布，而且很容易使用样本回答查询**。

11.1.2　拒绝抽样

迄今为止，您已经看到根据模型中的元素定义生成元素值的前向抽样过程。目前为止定义的过程没有考虑以条件或者约束形式出现的证据。如果没有任何证据，程序定义的是可能世界上的**先验分布**。所以，前向抽样过程从先验概率分布中生成样本。

尽管这对理解过程很有帮助，但是我们通常对回答有关**后验分布**的查询感兴趣，这种概率是根据证据进行调节之后得到的。您需要一种从后验分布中生成样本的手段。

实现上述目标的一种简单过程称作**拒绝抽样**。拒绝抽样的原理很简单，像前面一样使用前向抽样，但是拒绝任何与证据不符的样本。只有和证据保持一致的样本得以维持。

注意： 这里介绍的算法是一般数学概念的特例。本节介绍的拒绝抽样算法只适用于有条件但是没有软约束的概率程序。

拒绝抽样是第 4 章中您已经看到的根据证据进行调节的基本方法的结果。图 11-6 应该能够让您回想起这种方法。您从可能世界上的先验概率入手，在观察到证据时，“删去”与证据不相符的可能世界，为其指定概率 0。然后规格化所得的概率，得到后验分布。

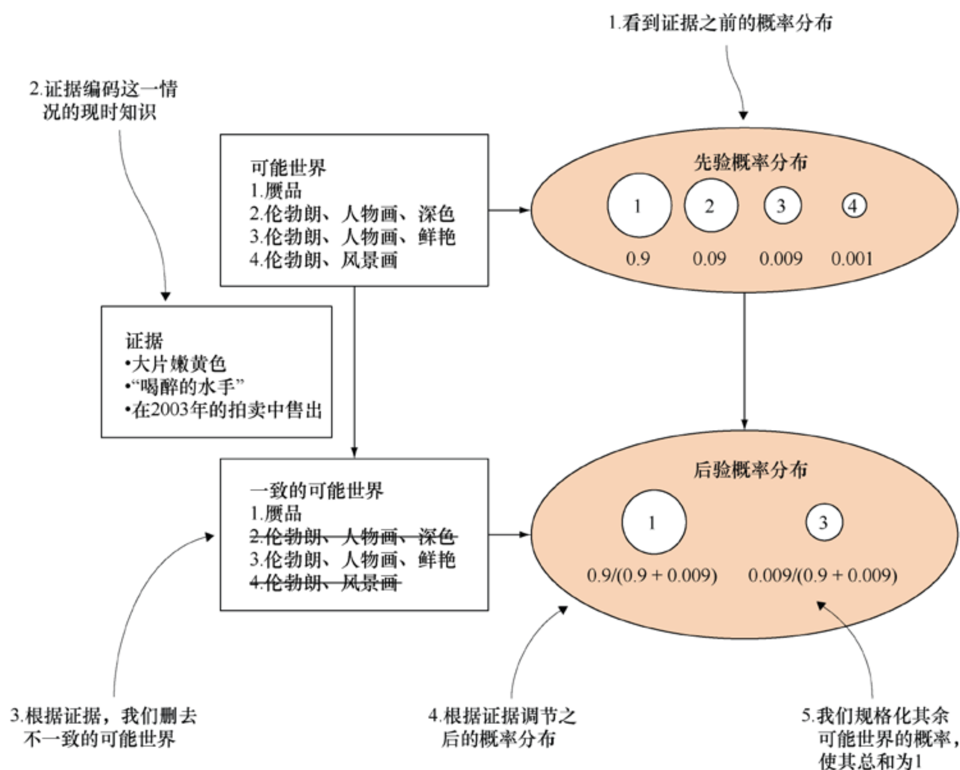


图 11-6　重现图 4-4：根据证据调节的过程

拒绝抽样使用相同的原理，删去和证据不符的可能世界。这通过删除与证据不符的所有样本来实现。剩余样本代表着根据证据调节之后的后验概率。然后，可以用这些剩余的样本回答任何查询。

我们来考虑表 11-1 中示例使用的同一个程序，但是这次增加一些证据。下面是修改后的程序：

```
val rembrandt1 = Flip(0.4)
val rembrandt2 = Flip(0.2)
val samePainter = rembrandt1 === rembrandt2
rembrandt2.observe(false)
```

表 11-2 的内容和表 11-1 一样，但是删除了与观测值不符的样本。剩下的样本代表了后验概率分布。

表 11-2　Flip(0.2)元素观测值为 true 的拒绝抽样。任何与此观测值不符的样本都被拒绝

元　　素	第 1 个样本	第 2 个样本	第 3 个样本	第 4 个样本
rembrandt1	true	false	true	false
rembrandt2	false	false	true	false
samePainter	false	true	true	true

表 11-2 说明了拒绝抽样的基本概念，但是真正的拒绝抽样算法有两处优化。首先，保存最终将被拒绝的样本不能高效地利用内存。特别是如果大部分样本被拒绝，可能导致您在得到后验分布中的一组样本之前就耗尽内存。因此，与证据不符的样本在创建之后立刻被抛弃，不予保存。

其次，在检查样本与证据是否一致之前，没有必要生成整个样本。可以在生成的元素值与元素相关证据不符时立刻抛弃样本。在表 11-2 中的第 3 个样本，一旦生成 rembrandt2 的值 true，您就知道该样本将被拒绝，因此没有必要生成 samePainter。与等到整个样本生成之后再拒绝相比，这可以大大地节约时间。

下面是拒绝抽样的伪代码。这些代码多次尝试抽样，直到生成与证据一致的样本。

1．假设 O 是变量的拓扑顺序。

2．对于顺序 O 中的每个变量 V：

a）假设 Par 是 V 的父变量。

b）假设 x_{Par} 是之前生成的 Par 值。

c）从 $P(V \mid Par = x_{Par})$中提取 x_V。

d）如果 x_V 与关于 V 的证据不符，重复步骤 2。（该样本立即被拒绝）

3．返回 **x**。

下面是拒绝抽样的必知事项。

- 拒绝抽样的好处是从根据证据调节的后验概率分布中提取的样本不会被拒绝。而且，使用的样本越多，可以预期样本分布越接近于真实分布。这和前向抽样

相同，但是现在考虑了证据。

■ 拒绝抽样的缺点是大部分样本可能被拒绝。这样，大部分工作都浪费了，生成合适的样本集需要花费很长的时间。一般来说，样本被接受的概率等于证据的概率。所以，如果证据的概率很低，大部分样本将被拒绝。

可以估算证据的概率吗？假定您投掷一枚公平硬币 10 次。可能的结果数量为 2^{10}。所以，10 次掷币的任何一个观测值的概率为 $1/(2^{10})$。如果投掷 20 次，概率则为 $1/(2^{20})$。这意味着，如果在这个例子中使用拒绝抽样，那么只有 $1/(2^{20})$的样本被接受——这是一个很小的数量。一般来说，证据的概率随着拥有数据的变量数量增加而呈指数式下降。也就是说，使用拒绝抽样生成一个好的样本集所需的工作量随着证据变量的增加而呈指数式增加。您可能认为拒绝抽样快速拒绝样本的能力可以缓解这一问题。遗憾的是，这一能力产生的时间节约仅与证据变量呈线性关系，无法弥补指数式上升的代价。

因此，尽管对于阐述一般原理很有益，但是拒绝抽样通常不被视为实用的算法。此外，拒绝抽样仅能处理 Figaro 的条件，而无法处理更通用的约束。在下面两个小节中，您将看到两种实用的算法：重要性抽样和马尔科夫链蒙特卡洛算法。重要性抽样和拒绝抽样类似，但是使用更巧妙的方法考虑证据，而马尔科夫链蒙特卡洛算法使用完全不同的抽样方法。

11.2 重要性抽样

重要性抽样类似于拒绝抽样。实际上，当重要性抽样遇到不能满足的条件时，它将和拒绝抽样一样拒绝该样本。但是两者之间存在两个重大的差别。首先，除了条件之外，Figaro 提供了约束，约束不仅有真或假两种取值，而是为每个状态指定一个实数。拒绝抽样无法处理这种约束，因为样本不会完全与证据不一致，只是概率较低而已。重要性抽样可以处理不会造成拒绝的约束。其次，在合适的情况下，重要性抽样可以将条件转换为其他变量上的约束，避免因为条件而拒绝样本。

注意：重要性抽样是比我这里所描述的更加通用的一种算法框架。我将焦点放在 Figaro 使用的重要性抽样变种上。

在描述重要性抽样之前，我将详细地介绍 Figaro 条件和约束的含义。

■ **硬条件**是从元素值到布尔值的函数。它规定了某个值拥有正概率所必须满足的属性。准确地说，任何元素值不满足该条件的可能世界概率将为 0。因此，可以这样看待条件：如果元素不满足条件，将可能世界的概率乘以 0，如果满足条件则乘以 1。

■ **软约束**是从元素值到一个实数的函数。我曾经说过，约束可以解读为“其他条件都相同”的陈述。例如，假定您有一个元素的约束，如果该元素为 true 则

返回 1.0，为 false 时返回 0.5。假定两个可能世界的差别仅是这个元素的值，在其他条件相同的情况下，该可能世界中这个元素为 true 的概率两倍于 false 的概率。

还有一个更为精确的定义，考虑具有约束 C 的元素 E。忽略该约束，任何可能世界将有某一概率 p_0。假定该可能世界中的元素值为 e。考虑约束后该元素的非规格化概率是 $p_0 \times C(e)$。我称之为**非规格化**是因为当您这样解读约束时，这些值的总和通常不为 1，所以要将它们当成概率必须进行规格化。

做此简介之后，您就已经为研究重要性抽样的工作原理做好了准备。

11.2.1　重要性抽样的工作方式

在重要性抽样的主要原理中，与前面介绍的算法不同的部分是使用了**加权样本**：每个样本都不是完整的样本，仅根据其权重代表部分样本。然后，可能世界上的概率分布由样本的权重定义。我将在接下来解释其工作原理，然后说明重要性抽样如何避免拒绝。

加权样本

在重要性抽样中，每个样本与一个**权重**关联。权重基于样本的条件和约束值。当您遇到一个约束时，必须将可能世界的概率乘以约束值。这通过将样本权重乘以约束值实现。最终，权重的值将是所有约束值的乘积。

我们对表 11-2 使用的程序做如下修改，观察上述过程：

```
val rembrandt1 = Flip(0.4)
val rembrandt2 = Flip(0.2)
val samePainter = rembrandt1 === rembrandt2
rembrandt2.addConstraint((b: Boolean) => if (b) 0.1 else 1.0)
```

在这个程序中，您已经用软约束代替了硬条件。如果 rembrandt2 为 false，约束的值为 1.0；否则，约束值为 0.1。当您生成表 11-1 和表 11-2 中的样本时，为每个样本关联一个权重，权重值等于约束值。这样得到的加权样本如表 11-3 所示。

表 11-3　加权样本。每个样本的权重是 Flip(0.2)上的约束值

元　　素	第 1 个样本	第 2 个样本	第 3 个样本	第 4 个样本
rembrandt1	true	false	true	false
rembrandt2	false	false	true	false
samePainter	false	true	true	true
权重	1.0	1.0	0.1	1.0

一组加权样本定义了可能世界的概率分布。想象随机选择样本的过程，其中选择某个样本的概率与其权重成正比。为了得到选择某个样本的概率，您简单地将其权重除以

所有权重的总和（表 11-4 中的 4 个样本权重总和为 3.1）。例如，选择第一个样本的概率为 1/3.1，而选择第 3 个样本的概率为 0.1/3.1。某个可能世界的概率为选择与其相符的样本的概率总和。所以，对应于第 2 个和第 4 个样本的可能世界概率为 2.0/3.1。综上所述，表 11-3 中的加权样本定义了表 11-4 中的后验概率分布。

表 11-4 重要性抽样产生的一组加权样本定义的后验概率分布。可能世界的概率和与之相符的样本权重总和成正比

Flip(0.4)	Flip(0.2)	Flip(0.4) === Flip(0.2)	概率
true	false	false	1.0 / 3.1 = 0.3226
false	false	true	2.0 / 3.1 = 0.6452
true	true	true	0.1 / 3.1 = 0.0322

因为加权样本定义了一个概率分布，它们可以用于回答查询，完成这一工作很简单。例如，假定您想知道 x（Flip(0.4)元素）为 true 的概率。x 为 true 的样本为第一个和第三个样本。它们的总权重为 1.1。将总权重除以所有样本的总权重（3.1），可以得到估算的 P(x = true) = 1.1 / 3.1 = 0.3548。

在重要性抽样中，当您将元素乘以证据时，使用的是每个证据元素的约束值。例如，假定您将 x 上的第二个约束添加到我们的程序中：

```
val rembrandt1 = Flip(0.4)
val rembrandt2 = Flip(0.2)
val samePainter = rembrandt1 === rembrandt2
rembrandt2.addConstraint((b: Boolean) => if (b) 0.1 else 0.9)
rembrandt1.addConstraint((b: Boolean) => if (b) 0.8 else 0.3)
```

rembrandt1 为 true 且 rembrandt2 为 true 的可能世界权重等于 0.8 × 0.1 = 0.08。

正如前向抽样，需要强调的是，这不是精确的后验概率，只是根据生成的样本进行的估算。重要性抽样和前向抽样有相同的属性。

- 平均来说，样本分布等于真实分布。
- 使用的样本越多，可以预期样本分布越接近于真实分布。

因此，重要性抽样是概率编程中很好的一种近似推理方法。

避免拒绝

在前一个例子中，证据以软约束的方式指定。这是一件好事，因为重要性抽样的目标之一就是处理约束。但是如果您有一个条件，会发生什么？

条件等价于值为 0 和 1 的约束。例如，您可以断言如下证据

```
y.observe(false)
```

等价于

```
y.addConstraint((b: Boolean) => if (b) 0.0 else 1.0)
```

因为您知道如何在重要性抽样中处理约束，所以也可以处理条件。但是有一个陷阱。如果样本与硬条件不符，其权重将为 0，所以对于后验分布没有任何贡献。这等同于拒绝该样本。因为您知道样本的权重最终将为 0，可能直接拒绝以节省一些工作量。

现在，是时候看看基本重要性抽样算法的伪代码了。如下代码返回样本及其权重。

1．假设 O 是变量的拓扑顺序。

2．w ←1（权重初始化为 1；如果没有约束，最终约束也将为 1）。

3．对于顺序 O 中的每个变量 V：

a）假设 Par 是 V 的父变量。

b）假设 x_{Par} 是之前为 Par 生成的值。

c）从 $P(V \mid Par = x_{Par})$中提取 x_V。

d）如果 x_V 与 V 的条件不一致，转到 2。

e）w ←w *（适用于 X_v 的 V 上约束的乘积）。

4．返回(**x**, w)。

上述算法在硬约束上并不比拒绝抽样更好，我曾经说过后者可能是一个不实用的算法，即使只有一个样本，生成的时间也过长。即使不可能完全避免拒绝与条件不符的样本，也应该尽可能做到。这个问题的解决方案是通过程序试图“回推证据”，使后续元素上的条件变成之前元素上的软约束。完成这一目标的通用过程很复杂，但是我可以用如下例子说明：

```
val x = Beta(1, 1)
val y = Flip(x)
y.observe(false)
```

让我们来分析上述例子。首先，为 x 生成一个值。假定生成的值为 0.9。接下来，使用 Flip(0.9)为 y 生成值。该变量取值为 true 的概率是 0.9，为 false 的概率是 0.1。所以观测值满足条件的概率是 0.1。您不需要对 y 进行抽样就可以知道这一点；从 Flip 的定义立即可以看出。在得到 x 的样本 0.9 之后，您立刻就可以为这个样本指定权重 0.1，不需要对 y 进行抽样。类似地，如果 x 的样本为 0.3，观测值满足条件的概率是 0.7，所以您立刻可以为该样本指定权重 0.7。

一般来说，如果 x 的抽样得到值 p，该观测值的概率为 1-p。您可以引入 x 上的一个约束，模拟为样本指定权重 1-p 的过程。这个约束等于 1 减去 x 值。然后，您可以强制 y 为 false；您知道从该观测值可以看出，y 必然为 false，您已经考虑了这个观测值对 x 上的约束发生的影响。总结起来，您的程序等价于：

```
val x = Beta(1, 1)
x.addConstraint((d: Double) => 1 – d)
val y = Constant(false)
```

现在知道如何将硬条件转换为软约束了吧？结果是，您不会拒绝这个程序中的任何

样本，而是根据 x 的值为其设置一个权重。

注意： Figaro 没有包含为所有条件进行上述运算的通用规程，而是使用一组处理常见情况的简单启发方法。使用程序变换移动前面的条件和约束是一个活跃的研究领域，我们预测未来 Figaro 在这方面的能力将得到改善。

11.2.2　在 Figaro 中使用重要性抽样

在 Figaro 中使用重要性抽样很简单。Figaro 提供了重要性抽样的两种版本：一次性版本运行于给定的样本数量上，任意时间版本可以运行任意长的时间，在运行中同时得到查询的答案。使用这些算法必须导入 com.cra.figaro.algorithm.sampling.Importance。下面的代码在查询目标 rembrandt1 和 rembrandt2 的 10000 个样本上运行一次性的重要抽样算法，然后对结果发出查询：

```
val algorithm = Importance(10000, rembrandt1, rembrandt2)
algorithm.start()
println(algorithm.probability(rembrandt1, true))
println(algorithm.distribution(rembrandt2).toList)
algorithm.kill()
```

重要性抽样的任意时间版本在运行时收集越来越多的样本，得到越来越好的查询答案。不传递样本数量的整数参数而只传递查询目标，就可以使用任意时间版本。您通常可以在算法运行时进行其他工作。等待一段时间的简单方法是调用 Thread.sleep，该命令等待给定的毫秒数之后继续。

下面的代码运行任意时间重要性抽样 1 秒，打印查询答案，然后再运行 1 秒，再次打印答案：

```
val algorithm = Importance(x, y)
algorithm.start()
Thread.sleep(1000)
println(algorithm.probability(x, true))
println(algorithm.distribution(y).toList)
Thread.sleep(1000)
println(algorithm.probability(x, true))
println(algorithm.distribution(y).toList)
algorithm.kill()
```

任意时间算法在单独线程中运行。在结束时杀死任意时间算法以释放该线程很重要；否则它将继续使用系统资源。

如果想要停止算法工作，在以后恢复，可以使用 algorithm.stop() 和 algorithm.resume() 方法，这些方法的作用正如您的预期，例如，可以使用如下代码：

```
val algorithm = Importance(x, y)
algorithm.start()
Thread.sleep(1000)
```

```
algorithm.stop()
// Show an interactive visualization and wait for the user to be done
algorithm.resume()
Thread.sleep(1000)
algorithm.stop()
// Show an updated visualization, etc.
```

11.2.3 让重要性抽样为您工作

何时应该使用重要性抽样？使用这种算法通常有两个主要原因。

- 您的模型有连续变量。正如您在第 10 章中所看到的，虽然因子分解算法可以处理连续变量，但是它们通常必须枚举那些变量的少数值，因此不像处理范围较小的离散变量那么有效。如果想要考虑这些连续变量的整个范围，应该使用重要性抽样等抽样算法。

  具有连续变量的模型示例之一是预测不同产品线销售额的模型。销售额以美元计量，将其作为连续变量最为自然。如果使用因子分解算法，就必须将销售额归纳为相对少的数值，这可能无法得到您所需要的解。
- 您的模型结构可变。如果模型中的某些部分只在其他变量的特定值下存在，重要性抽样将仅生成与每个样本相关的变量。这方面的一个例子是第 8 章中介绍的饭店可变结构动态模型。任何时间步中在饭店就坐的人数都是可变的。在 8.2.4 小节中，使用了重要性抽样进行该模型的推理。

需要强调的是，重要性抽样并不是万能的，即使对于它适合的模型类型也是如此。问题是收集指定数量的加权样本和收集同样数量的常规未加权样本不同。如果使用重要性抽样收集 100 万个加权样本，其效果可能仅等同于拒绝抽样收集的 100 个样本。**有效样本数**的概念描述了与一组加权样本等价的常规样本数。

有效样本数很难精确描述，但是从直觉上说，样本权重的总和越小，有效样本数越小。一般来说，样本的平均权重等于以条件和约束表达的证据的概率。因此，证据的概率越低，预期的有效样本数就越小。这个问题类似于拒绝抽样，在拒绝抽样中，证据概率越低，预期的可接受样本就越少。

考虑到这一点，使重要性抽样有效工作的最佳方法是避免可能性极小的证据。例如，不使用不太可能满足的硬条件，这种条件会导致大量的拒绝样本，作为替代，使用条件不满足时较低但非零的约束值。如果您有许多这样的条件并且用约束替代它们，就会发现与大部分条件一致的样本将得到比和较少条件相符的样本更高的权重。这在许多情况下能够得出较好的查询答案。遗憾的是，虽然我可以直观地解释这一点，但是在实践中很难准确地控制。关键的条件是与更多条件相符的样本比与较少条件相符的样本更接近于真实的情况。接下来的补充材料中有一个例子。

示例：用重要性抽样破解密码

想象一下用概率推理破解密码。密码是一个用编码写成的句子，字母表中的每个字母都被不同的字母替代。您可以建立一个概率模型，随机选择替代每个字母的特定字母。您可能有两组条件：任何两个字母都不会由同一个字母替代；句子中的每个单词都必须是一个有效的英语单词。

对于拒绝抽样，您将随机抽样一组替换，然后检查它们是否满足条件。生成满足所有条件的一组替换的概率极小，所以生成一个样本的时间很长。但是如果将这些条件改成软约束，就可以生成满足大部分（但是并非全部）约束的替换。这些样本可以帮助您估算有用的后验分布。虽然无法生成完全正确的答案，但是可以从大部分高权重样本中发现，字母 E 被替换为字母 N。这能帮助您进行猜测，不断推进以解决谜题。这一过程和人类破解密码的过程有些类似。

重要性抽样的本质是：

- 这是一种适合于任何模型的通用算法。
- 当证据的概率较低时，因为有效样本数很小，该算法可能失败，但是您可以通过放松证据提供帮助。

11.2.4 重要性抽样的应用

您已经看到，在没有可能性极小的证据时，重要性抽样工作得更好。相应地，大部分重要性抽样的实际应用都是证据很少或者完全没有证据的问题。

通过模拟预测

当您完全没有证据（程序中没有条件或者约束）时，重要性抽样等同于 11.1.1 小节中介绍的最简单的前向抽样算法。前向抽样通过生成大量未来的可能运行，预测未来的发展。这样使用时，Figaro 可以作为一个有效的模拟系统。您可以使用 Figaro 的重要性抽样算法实现上述目的。您可以在许多相关应用（如选举、军事行动计划、经济制度和体育比赛预测）中以这种方式使用 Figaro。

例如，假定您想要模拟一个足球赛季，预测球队最后的排名。您可以创建一个依赖于比赛双方能力的单独比赛模型，并将其嵌入到由所有比赛组成的整个赛季的模型中。您的模型可以考虑球员伤病等问题。进行前向抽样时，您将模拟整个赛季的每场比赛，以提出最终的联赛排名表。使用这些模拟，可以计算球队赢得冠军的概率。这个模型需要考虑整个赛季的进程，因此十分复杂，抽样是进行这种推理的自然方法。

这样使用重要性抽样时，您通常拥有初始条件，如联赛中所有球队的初始实力。建立这些条件的模型有三种方法。

- **固定 Scala 值**——这种方法可以高效地推理，但是无法建立赛季中球队实力随机变化的模型。如果没有规划模型的变化，这可能是最佳的方法。
- **Figaro 常量（Constant）元素**——例如，如果您的球队初始实力为 0.96，可以使

用 Constant(0.96)元素表示其初始实力。然后，您可以在赛季的每个时点用不同的元素表示当时的实力。这种方法的好处是允许实力在赛季中变化，但是在赛季开始时没有必要引入关于实力的证据。常量元素只能有一个值，所以球队的初始实力始终为 0.96。这使得前向抽样可以高效工作。

- **具有不固定分布的 Figaro 元素**——如果您对初始条件不确定时可以这么做。例如，对于每个球队，您可能认为其实力分布于某个范围中——例如，您的球队实力可能在 0.94 和 0.98 之间，可以使用 Uniform(0.94, 0.98)元素。这样做的好处是没有必要选择一个特定值。但是如果您心中有一个特定值（如 0.96）,那么使用常量元素更好，而不是使用这种方法并将证据设置为 0.96。

用关于初始状态的证据预测

如果您不知道准确的初始状态，可能有某些相关的证据。例如，您可能相信球队的实力与上一赛季的最终排名有关，并用赛季之后转入和转出的球员进行调整。为了加入这些证据，您可以对前一赛季进行抽样，以赛季最终排名作为证据，推断前一赛季所有球队的实力。然后，根据球员的转入转出情况对它们的实力进行调整。最后，您可以使用这些实力作为初始条件模拟新赛季。下面是实现上述功能的 Figaro 代码骨架：

```
val lastYearsStrengths = Array.fill(Uniform(0, 1))
val lastYearsTable = playSoccerSeason(lastYearsStrengths)
lastYearsTable.observe(actualTable)
val thisYearsStrengths =
  lastYearsStrengths.map((strength: Element[Double]) => adjust(strength))
val thisYearsTable = playSoccerSeason(thisYearsStrengths)
println(Importance.probability(thisYearsTable, (t: Table) => myTeamTop(t)))
```

只要关于初始状态的证据概率不会太低，重要性抽样就是这一应用的出色候选。如果您观察了整个赛季的比赛结果，对于重要性抽样来说，证据就太多了，使用 MCMC 的效果更好。

您可以想象将这一推理扩展到多个赛季。也许，可以使用 3 个过去的赛季决定初始条件。在每个过去的赛季中，从赛季的初始实力开始，模拟该赛季的比赛，观察联赛排名，并根据球员的增删进行调整，得出下一赛季的实力。这些推理都可以用重要性抽样实现。

在这里，我所描述的是动态模型推理所用的粒子过滤算法的基本原理。在粒子过滤中，从一组表示系统在某个时点的当前状态的样本开始，使用重要性抽样考虑该时点的证据。粒子过滤增加了一个**重新抽样**步骤。您将在第 13 章中学习关于粒子过滤的所有知识，但是在这里我要指出，重要性抽样的主要应用之一就是粒子过滤。

对证据较少的复杂过程的推理

重要性抽样不仅用于预测未来，对于推理任务也很有效，这种任务的目标是根据查询变量的效果推断该变量值。通常，在产生结果的过程很复杂且证据变量不多时使用重

要性抽样。下面是一个例子。

社会化网络分析使用各种网络生成模型，如 Erdos-Renyi 模型和 Barabasi-Albert 优先连接模型。社会科学家们可能想要知道哪个模型更适合于某种观察到的网络。不同模型可能产生不同的统计数字，如节点最大邻接数量，或者两个节点间的平均距离。使用这些统计，可以推断可能已经使用了哪种网络生成过程。在 Figaro 中，不同的网络生成模型将返回 Element[Network]。统计将在这些元素之上运行，并返回 Element[Double]。下面是这一应用的代码骨架：

```
class Network { … }
def erdosRenyi: Element[Network] = …
def barabasiAlbert: Element[Network] = …
def maxNeighbors(n: Network) = …
def averageDistance(n: Network) = …
val er = erdosRenyi()
val ba = barabasiAlbert()
val myNetwork = discrete.Uniform(er, ba)
val mn = myNetwork.map(maxNeighbors)
val ad = myNetwork.map(averageDistance)
mn.observe(7)
ad.addCondition((d: Double) => d > 0.31 && d < 0.35)
println(Importance.probability(myNetwork, er))
```

Apply(myNetwork, (n: Network) => maxNeighbors(n))的简写形式

为条件设定一个范围，避免它的可能性太小

重要性抽样在这种场合下很有效，因为证据仅在网络的摘要统计中。尽管不太可能生成特定的网络，但是许多网络的统计数字类似，所以摘要统计的概率不会太小。相反，如果您已经观察到精确的生成网络，任何一种网络生成方法恰好生成观察到的网络的可能性都很小，所以重要性抽样不是很有效。

从上述讨论可以看出，重要性抽样仅在证据变量不太多且证据的概率不太小的情况下实用。现实世界中的许多情况不满足这些条件，所以需要不同的算法。一般来说，这个算法就是下一节要讨论的 MCMC。

11.3　马尔科夫链蒙特卡洛抽样

马尔科夫链蒙特卡洛（MCMC）算法解决重要性抽样的根本局限性。在重要性抽样中，生成一个有高权重的“好”样本可能需要很长的时间。在生成一个样本之后，您必须重新开始，生成下一个样本。MCMC 的主要原理是在每个样本中不重新开始，而是从前一个样本停止的地方开始算法的每一步。这有两个主要的好处。

- MCMC 可以更快地得到有较高概率的样本。
- 找到高概率的样本之后，MCMC 倾向于停留在高概率样本的区域中。

MCMC 是灵活、强大的算法，我们来看看它的工作方式，然后考虑实践中的问题。

11.3.1 MCMC 的工作方式

我们使用如下的示例程序说明 MCMC 的工作方式：

```
val x = Normal(0.75, 0.2)
val y = Normal(0.4, 0.2)
x.setCondition((d: Double) => d > 0 && d < 1)
y.setCondition((d: Double) => d > 0 && d < 1)
val pair = ^^(x, y)
println(MetropolisHastings.probability(pair,
          (xy: (Double, Double)) => xy._1 > 0.5 && xy._2 > 0.5))
```

这个程序定义两个正态分布随机数，并规定了确保其值在 0 和 1 之间的条件。然后，创建由两个数字组成的配对。最后，使用 Metropolis-Hastings 算法（Figaro 的 MCMC 算法）计算两个数均大于 0.5 的概率。

图 11-7 展示了 MCMC 的工作方式。图中显示一组可能世界（x 和 y 取值为 0 和 1 之间的正方形区域）。可能世界上有某种概率分布；圆形区域表示有高概率的可能世界。MCMC 算法经历一个状态序列。在每个时间步中，它随机地迁移到一个新状态。每个步骤都是随机的，但是它倾向于转移到概率较高的状态中。最终，算法通常到达具有高概率的状态。

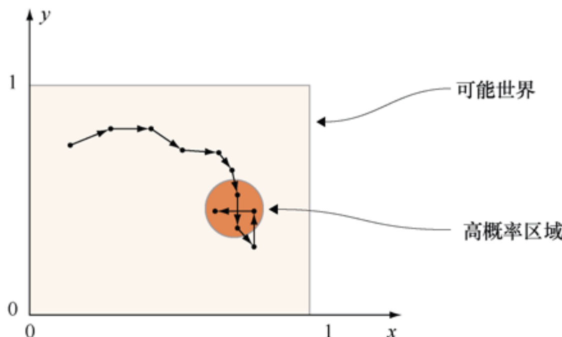


图 11-7 MCMC 过程：算法随机地在一系列状态中转移，逐渐接近高概率的状态

为什么 MCMC 有效

让我们尝试理解 MCMC 为什么有效。需要理解的关键点有三个，如果理解了这三点的逻辑，就掌握了该算法的精髓。

顾名思义，MCMC 依赖第 8 章中介绍的马尔科夫链理论。您应该记得，马尔科夫链是经历一系列状态的动态系统的一种模型。在每个时间步中，系统根据某种条件概率

分布，从当前状态迁移到下一个状态。很容易看出 MCMC 算法定义马尔科夫链的原因：它经历系统的一系列状态，在每个时间点根据事先定义的过程从一个状态转移到下一个状态。图 11-8 展示了这种马尔科夫链，使用的框图类型与第 8 章相同。

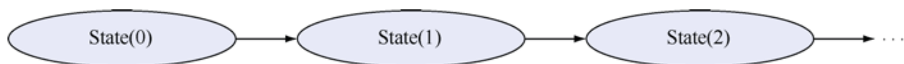


图 11-8　展示 MCMC 所经历状态的马尔科夫链

现在，让我们完成思想上的一次飞跃，从**状态序列**转向**分布序列**。逻辑如下：在马尔科夫链的任何一次运行中，系统经历一个状态序列。在任何时点，系统可能处于一些状态中的任何一个。当前系统状态总是有一个概率分布；在任何时点，都有针对该时点的一个概率分布。换言之，马尔科夫链定义一个分布序列，每个时点对应一个分布。

让我们来定义这个分布序列。根据任何时点的当前分布，马尔科夫链的迁移模型定义了下一个分布。在 Figaro 中，这可以用 Chain 实现，如：

```
val nextDistribution = Chain(currentDistribution, (s: State) =>
transition(s))
```

下一个分布由如下生成过程定义：从当前分布中提取的某个状态开始，应用迁移函数以得到下一个状态。因此，您可以得到如下属性。

- **要点** 1——从给定的初始分布开始，马尔科夫链定义了系统状态的概率分布序列。

现在介绍 MCMC 的微妙之处，以及它发生作用的原因：

- **要点** 2——对于满足某个数学条件的任何马尔科夫链，后续状态的分布收敛于单一分布。

这个分布被称作马尔科夫链的**平稳分布**。上述过程如图 11-9 所示。该图中最重要的一点是，阴影区域显示的不是 x 和 y 的可能状态，而是 x 和 y 的概率分布空间。这个空间中的每个点都是一个完整的分布，而不是 x 和 y 的一个值。

马尔科夫链从特定的初始分布开始。在 MCMC 中，初始分布由为算法选择系统初始状态的过程确定。在我们的示例程序中，这个过程选择 x 和 y 的初始值。例如，这个过程可能将 x 和 y 都设置为 0.5；这将定义一个概率分布，使该状态的概率为 1。

图中的箭头说明了从某个时点的分布转移到下一分布的过程；同样，这不是状态间的迁移，而是状态分布之间的转移。如图所示，不管您从哪一个初始分布开始，分布序列都收敛于一个极限——平稳分布。平稳分布可能永远无法达到，但是经过足够的步骤，可以接近到您所预期的距离。而且，在算法收敛到接近平稳分布之后，它将保持接近。

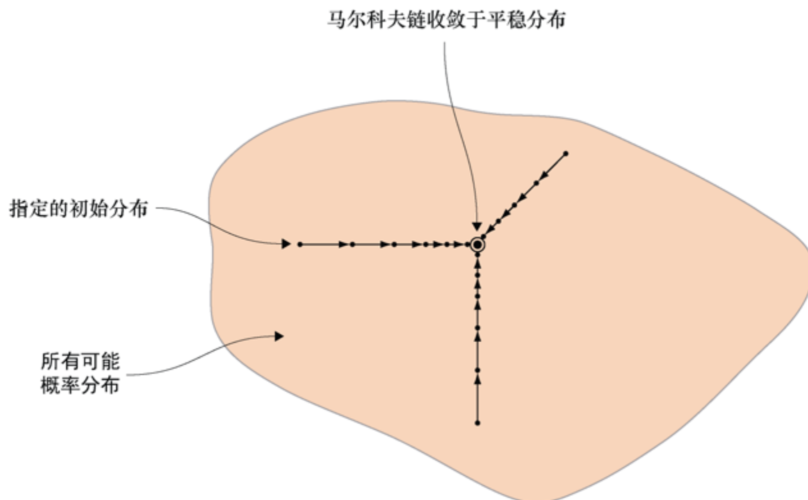


图 11-9 马尔科夫链定义的状态分布随时间推移而收敛，阴影区域展示了状态的所有可能概率分布。每条路径展示了从不同初始分布开始的分布序列。不管从哪个初始分布开始，过程都收敛于同一个平稳分布

MCMC 的最后一个要点是：

- **要点 3**——如果设计马尔科夫链使平稳分布成为您打算从中抽样的后验分布，那么从这个平稳分布中抽样，就接近于从后验分布中抽样。

MCMC 算法的定义

MCMC 算法的精髓在于，从初始状态开始，使用马尔科夫链反复转移到某个新状态。在较长的一段时间之后，就可以知道当前状态分布接近于您希望从中抽样的后验分布。那时，您就可以记录一个样本。记录样本之前经过的马尔科夫链步数称作**老化时间**。老化时间越长，距离真实的后验分布越近。另一方面，老化时间越长，MCMC 记录一个样本所需的时间越长。

在 11.3 小节开始时我曾经提到，MCMC 的一个好处是，找到高概率状态之后，它就从该状态开始，而不是从头开始。在 MCMC 中，您从不回到初始状态，而是继续使用马尔科夫链在状态之间迁移。原则上，在收集一个样本之后，就已经知道接近真实后验分布了，所以您应该可以在每一步中收集到一个样本。

但是，这里有一个微妙之处：MCMC 后续步骤的状态并不是相互独立的。考虑我们的示例程序。假定在某个时间步上，x 值为 0.9。如果马尔科夫链对 x 的移动量很小，下一步的 x 值将接近于 0.9。它将与前一个值强相关。这可能导致查询答案出现偏差。因此，您可以指定所收集样本之间的**间隔**，确保样本相互之间的相关度较小。但是，在实践中人们发现，使用某个间隔并不一定能够改善 MCMC 的性能，因为这会造成样本数的显著减少。

图 11-10 展示了使用老化和间隔的 MCMC 抽样程序的进展情况。可以看到，使用

老化确保收集到的第一个样本已经处于高概率区域，但是没有任何保证。您只能猜测老化所需时间，希望老化的效果足够大。在图中还可以看到，使用间隔 2 确保后续的样本距离较远，这意味着它们有一定的独立性。

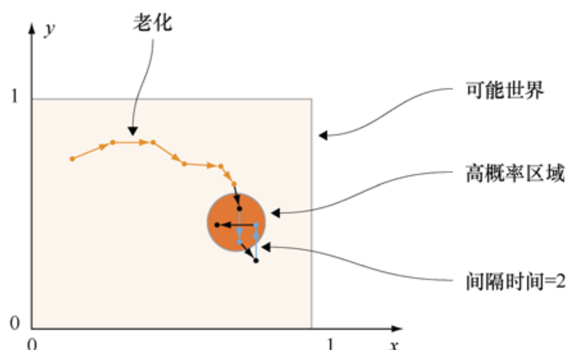


图 11-10 使用老化和时间间隔的 MCMC 抽样程序进程。老化样本被跳过，因为间隔时间为 2，提取每两个样本中的第二个

下面是 MCMC 算法较为正式的规格说明。这一算法有 3 个参数：样本数量、老化时间和间隔时间。

1. 从马尔科夫链的初始分布提取一个初始状态。
2. 老化阶段——在老化时间内重复如下操作。
a）用迁移模型迁移到一个新状态。
3. 将当前状态记录为第一个样本。
4. 抽样阶段——重复如下操作直到收集预定数量的样本。
a）在间隔时间内重复如下操作。
i. 用迁移模型迁移到新状态。
b）将当前状态记录为一个样本。

11.3.2 Figaro 的 MCMC 算法：Metropolis-Hastings 算法

MCMC 的要点 3 指出，马尔科夫链的设计使得平稳分布成为您从中提取样本的后验分布，由此得到算法从后验分布中抽样的基础。设计具备这种属性的马尔科夫链有许多种方法。Figaro 的 MCMC 算法使用的方法基于所谓的 Metropolis-Hastings（MH）算法。MH 因其普遍的适用性而得到广泛应用，这一特性也使它适合于概率编程，它有能力表达许多种模型。

Metropolis-Hastings 的基本思路是使用迁移模型有如下两个步骤的马尔科夫链。

1. 根据当前状态（模型中变量的一组值），提出新的状态。这个新状态根据提议的

分布选择，该分布是在给定当前状态下，下一个状态的概率分布。提议的分布取决于当前状态；例如，它可能与当前状态只有很小的距离。MH 算法的设计艺术中，大部分是对提议分布的选择。

2．根据接受概率选择接受新状态还是保持当前状态。接受概率取决于提议分布的概率以及新状态与当前状态的相对概率。

在 Figaro 中，提议分布通过使用**提议方案**（proposal scheme）指定。有一个默认的提议方案，以及用于指定自定义提议方案的 API。默认提议方案的工作方式如下。

1．随机选择一个非确定性元素。非确定性意味着在参数已知的情况下该元素值也是未知的；例如，Flip 和 Normal 是非确定性的，而 Constant 和 If 不是。

2．提出所选元素的一个新值。

3．根据选中的元素更新元素值。

在默认方案中，您无法控制第 1 步中选择的元素；它从非确定元素中均匀选取。在自定义提议方案中，您可以提议多个元素，也可以控制提议的元素。在某些情况下，默认方案工作得很好，通常值得首先尝试。在其他情况下，您需要设计自己的自定义方案。自定义提议方案在 11.4.1 小节中讨论。

在 Figaro 中创建一个 Metropolis-Hastings 算法

Figaro 提供以不同参数配置创建 MH 实例的手段。最常见的方法是指定您想要的样本数量、提议方案和查询目标。例如如下语句：

```
MetropolisHastings(100000, ProposalScheme.default, x, y)
```

上述语句指定使用默认提议方案，取得 10 万个样本，以获得 x 和 y 的近似后验分布。这是该算法的一次性版本，它一直运行到完成指定数量的样本。该算法还有任意时间版本，在这种版本中不指定样本数量，可以运行任意时长；例如：

```
MetropolisHastings(ProposalScheme.default, x, y)
```

在该算法的这些版本中没有老化参数，所以 MH 立即开始收集样本。时间间隔为 1：每个状态都被收集为一个样本。您也可以提供可选的老化和时间间隔参数。例如，用如下语句可以指定老化参数为 1000，总共取得 10 万个样本：

```
MetropolisHastings(100000, ProposalScheme.default, 1000, x, y)
```

在这个例子中，时间间隔为 1。如您所见，可以指定老化参数和默认间隔 1。指定老化参数很常见，因为您希望避免从马尔科夫链定义的早期分布中抽样，这些分布与真实分布可能相去甚远。但是在大部分情况下，不需要担心时间间隔，可以保留默认值 1。如果确实想要指定时间间隔，就必须同时指定老化时间。老化参数先出现，随后是时间间隔。例如，可以使用如下代码指定老化时间 1000 和时间间隔 10：

```
MetropolisHastings(100000, ProposalScheme.default, 1000, 10, x, y)
```

也可以使用上述所有变种的任意时间版本。图 11-11 展示了 MH 构造程序的结构，包括所有可选参数。

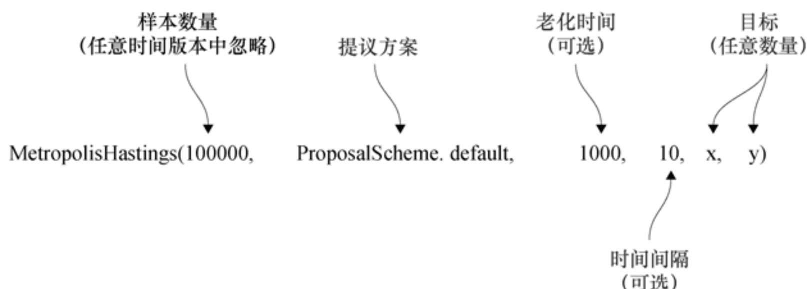


图 11-11　Figaro 中的 MH 构造程序结构

MH 的属性

MH 因其普遍适用性而得到广泛的应用。它的主要优势是不在每次迭代中随机抽样新状态，而是由抽样过程引导算法，在高概率状态中抽样。同样，在找到高概率区域之后，MH 倾向于停留在该区域。这样，在重要性抽样因为证据概率太低而无法正常工作的场合下，MH 仍然是可行的。

下面我们举一个包含两个变量（x 和 y）的简单例子，帮助您理解为什么 MH 能够找出高概率状态，而重要性抽样不能。假定您在 x 和 y 上有一个难以满足的约束。具体地说，假定 x 和 y 上的约束只在狭窄范围内取得高值，当您从它们的先验概率分布中抽样时，每 10 次抽样中只有 1 次能够得到高值，在其他 9 次中，约束值很低。使用重要性抽样，您每次都将从先验分布中生成新的 x 和 y，所以两个约束都取得高值的概率只有 1/100。这样，得到一个可用样本需要进行重要性抽样的 100 次迭代。

另一方面，让我们想象一下可以单独提出 x 或者 y 的 MH 过程。平均起来，在对 x 或者 y 的 10 次提议之后，其中一次的约束值将会较高，假定是变量 x。当 MH 之后再次提议 x 时，它几乎总是能够拒绝向约束值低的方向移动。所以 x 几乎总是停留在高概率区域。同时，对 y 再进行几次提议之后，它的约束也将取得高值。

现在想象同一个情况，但是这次有 100 个变量。对于重要性抽样，生成所有约束取高值的样本的概率为 $1/10^{100}$，这意味着实际上根本不可能。但是 MH 可以一次将一个变量移到高概率区域，最终找到所有变量都有高约束值的状态。

上述优势使 MH 对于许多应用来说都很有吸引力，但是它也有缺点。**主要的缺点是它可能是一个慢速算法**。如果没有指定样本之间的间隔，您通常必须收集比重要性抽样多得多的样本，因为样本之间不是相互独立的。根据具体的问题，您需要的样本数量可能比重要性抽样高出好几个数量级。如果决定使用间隔，间隔必须足够大才能使样本相互独立，这也就相应地加长了生成样本的时间。而且，老化和靠近高概率区域的时间也

可能很长。因此，在证据概率不是很小的情况下，重要性抽样是更好的选择。

MH 的第二个缺点是其行为可能难以理解、预测和控制。MH 的效率依赖于好的提议分布的定义。直观地说，您希望算法较快地转移到状态空间，这样找到高概率区域不需要花费很长的时间，而找到高概率区域之后就不会很快离开。这些属性对提议分布很敏感；设计具备上述属性的提议分布可能很难，特定提议分布的表现也难以预测。而且，很容易意外地定义不能完全探索状态空间的提议分布。我将在下一节对此进一步说明。

MH 的行为难以预测还产生了一个副作用：难以知道老化时间和总的样本数。如果您不知道达到高概率区域需要多长时间，如何知道何时停止老化？如果不知道从一个状态到几乎与之独立但仍留在高概率区域的另一个状态需要多长时间，如何知道取得的样本数量？因此，您通常需要长时间运行 MH，观察答案是否合理。一种选择是多次运行以观察不同运行中得到的答案是否相似；如果不相似，那么取得的样本数可能不足。

因此，尽管 MH 有很强的能力和普遍性，但是我常常将其视为最后手段。如果其他算法都不可行，尝试 MH，但是一定要了解，使其很好地工作可能需要更多的精力。下一节介绍帮助 MH 更好工作的技术。

11.4 让 MH 更好地工作

本节介绍使 MH 更好地工作的技术。我将使用一个具有挑战性的问题，如果不使用这些技术，该问题就难以解决。

想象您试图预测奥斯卡最佳演员奖的获得者。有一组候选人，每个候选人都是在某部电影中出演的演员。根据您的模型，是否获奖取决于演员的名望和电影的质量。

我们在 Figaro 中对该模型的第一次尝试是这样的：

```
class Actor {
  val famous = Flip(0.1)
}

class Movie {
  val quality = Select(0.3 -> 'low, 0.5 -> 'medium, 0.2 -> 'high)
}

class Appearance(val actor: Actor, val movie: Movie) {
  val award: Element[Boolean] =
    CPD(movie.quality, actor.famous,
        ('low, false) -> Flip(0.001),
        ('low, true) -> Flip(0.01),
        ('medium, false) -> Flip(0.01),
        ('medium, true) -> Flip(0.05),
        ('high, false) -> Flip(0.05),
        ('high, true) -> Flip(0.2))
}
```

上述代码使用了第 7 章介绍的面向对象建模模式，应该是不言自明的。但是这段代码有些问题。没有任何条件能阻止不同演出的获奖属性取得 true 值。所以，在任何可能世界中，都有任意数量的演出可能获奖。您当然知道情况不是这样的，所以添加一个条件，强制只有某次演出获奖，实现如下：

```
def uniqueAwardCondition(awards: List[Boolean]) = {
  awards.count(b => b)== 1
}
val allAwards: Element[List[Boolean]] =
  Inject(appearances.map(_.award): _*)
allAwards.setCondition(uniqueAwardCondition)
```

检查获奖者数量是否为 1

创建一个元素,其值是表示获奖的布尔值列表，从一个演出数组中收集

设置这个元素的条件，强制只有一个人获奖

为了完成情况的描述，必须创建演员和电影的实例，并用演出将它们联系起来。在实践中，演员、电影和演出可以从数据库中创建。在本书代码库中，我随机创建演出，代码如下：

```
val numActors = 200
val numMovies = 100
val numAppearances = 300

val actors = Array.fill(numActors)(new Actor)
val movies = Array.fill(numMovies)(new Movie)
val appearances =
   Array.fill(numAppearances)(new Appearance(
       actors(random.nextInt(numActors)),
       movies(random.nextInt(numMovies))))
```

遗憾的是，如果您直接将 MH 应用到这个问题，它完全无法工作。问题在于：在具有正概率的可能世界空间中，只有一次演出获奖。您希望 MH 探索具有高概率的可能世界空间。如果您从一次演出获奖的可能世界开始，打算转移到另一个状态，在该状态中这次演出没有获奖，而由另一次演出获奖。图 11-12 说明这种情况不会发生的原因。该图展示了两位演员、两部电影和两次演出的贝叶斯网络，但是不管数量多大都存在相同的问题。在默认的提议方案下，提议应该改变某位演员的声望、某部电影的质量或者某次演出的获奖状态。更改演员声望或者电影的质量都没有问题。但是，如果当前状态满足只有一次演出获奖的条件，更改某次演出的获奖状态就必然会违反这个条件。如果更改目前获奖的演出，就将转移到没有任何演出获奖的状态；如果更改当前没有获奖的演出，就会转移到两次演出都获奖的状态。在任何一种情况下，新状态都违反了上述条件，将被拒绝。没有办法在一个 MH 步骤中将获奖状态从某一次演出切换到另一次演出。

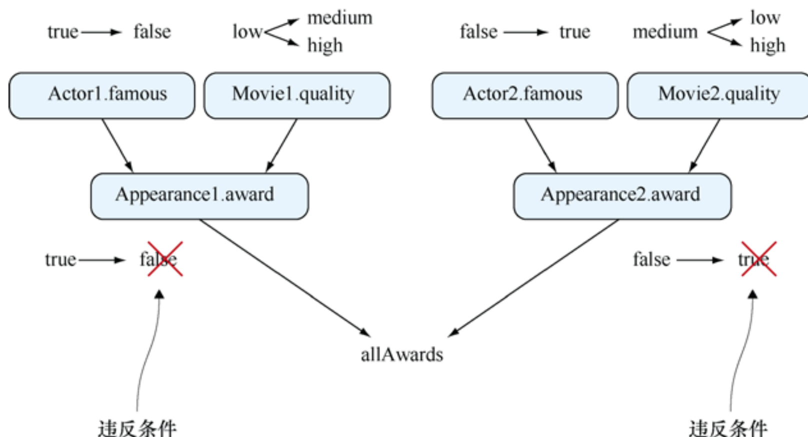


图 11-12　MH 直接应用到演员和电影示例中时可能的单步提议。变量以圆角矩形表示。对于每个变量，当前值在箭头的左侧，可能的替代值在右侧。在默认提议方案中，每一步 MH 都可以将任何一个变量从当前值更改为不同的值。两个值不能同时更改。因此，任何改变获奖变量值的提议都将违反唯一获奖条件，所以不能被接受

如何让 MH 能够用于解决这个问题？有两种主要的思路。第一种是使用不同于默认方案的提议方案。例如，自定义方案可以提议一次更改两个元素。第二种思路是避免不能违反的硬条件。我们的例子中的一个问题是必须绝对满足唯一获奖条件，提议才能被接受。如果您可以放松这一限制，在某些时候接受 0 或者 2 个获奖演出的提议，MH 算法有时就能更改获奖变量。下面的两个小节将详细介绍这两种思路。

11.4.1　自定义提议

第一个问题是您使用了默认提议方案。Figaro 提供了描述自定义方案的语言。您应该记得，默认提议方案随机选择一个非确定性元素，而自定义提议方案可以让您提议多个元素，控制提议的元素是哪一些。特别是，自定义提议方案可以帮助您：

- 依次提议多个元素。
- 在多个备选元素中选择。
- 从一个元素开始，根据该元素值决定如何继续。

下面搭配的补充材料提供了上述功能的实现细节。现在我将介绍如何为演员与电影示例创建一个自定义提议方案。在本例中，您希望能够同时提议两次演出。Figaro 使用如下结构依次提议多个元素：

```
ProposalScheme(element1, element2, …)
```

因此，如果 appearances 是 Appearance 类的实例数组，长度为 numAppearances，可以使用如下代码创建一个提议方案，提议两个随机演出的 award 元素:

```
ProposalScheme(
  appearances(random.nextInt(numAppearances)).award,
  appearances(random.nextInt(numAppearances)).award)
```

有时候，您还希望提议演员的声望或者电影的质量。为了得到所需的效果，您可以使用 DisjointScheme，让算法在多个提议方案中选择，为每个选择指定一个概率。在我们的例子中，创建一个 Scala 值 scheme，包含您想要的 DisjointScheme:

```
val scheme: ProposalScheme = {
  DisjointScheme(
    (0.5, () =>
      ProposalScheme(appearances(random.nextInt(numAppearances)).award,
                     appearances(random.nextInt(numAppearances)).award)),
    (0.25, () =>
      ProposalScheme(actors(random.nextInt(numActors)).famous)),
    (0.25, () =>
      ProposalScheme(movies(random.nextInt(numMovies)).quality)))
}
```

现在，您已经有了所需的提议方案，可以提议两次随机选择的演出的获奖情况（可能导致一次切换）或者随机选择的演员声望、随机选择的电影质量。

提议方案概述

Figaro 提供了一些定义提议方案的语言。除了默认方案之外，您还可以：

- **在多个备选方案中选择**——假定您想要在多个提议方案中随机选择，每种方案有不同的概率。DisjointScheme(probability$_1$ -> schemeFunction$_1$, ..., probability$_n$-> schemeFunction$_n$)可以实现这一功能。每个方案函数都没有参数，返回一个提议方案。在演员和电影示例中，() => ProposalScheme(actors(random.nextInt(numActors)).famous) 是方案函数的一个例子。DisjointScheme 以相应的概率选择其中一个方案函数，然后应用方案函数得出所要使用的提议方案。
- **依次提议多个元素**——ProposalScheme(element$_1$, ..., element$_n$)将依次提议 element$_1$ 到 element$_n$。因为您通常不希望在每次 MH 迭代中提出相同的元素，这种提议方案一般作为方案函数的结果使用，例如在 DisjointScheme 中使用的函数。在我们的例子中就是这么做的。
- **从一个元素开始，根据该元素值决定如何继续**——有时候，您希望提议一个元素，然后根据第一个元素选择的值决定如何继续提议。在演员和电影获奖时提议演员的声望和电影的质量就是一个例子。TypedScheme 用于这类情况，它有两个参数。第一个参数是返回第一个提议元素的函数，其类型是() => Element[T]，其中的 T 是第一个元素的值类型。TypedScheme

的第二个参数类型为 T => Option[ProposalScheme]。也就是说，该函数取得第一个元素的生成值（类型为 T），根据该值返回 None（表示提议应该停止）或者返回 Some[proposalScheme]（表示提议应该用给定的 proposalScheme 继续）。例如，考虑如下代码：

```
val appearance = appearances(random.nextInt(numAppearances))
TypedScheme(
  () => appearance.award,
  (b: Boolean) =>
    if (b) Some(ProposalScheme(appearance.actor.fame,
                               appearance.movie.quality))
    else None)
```

这个提议方案随机选择一次演出，首先提议该演出的获奖情况。如果 appearance.award 的新值为 true，则继续依次提议 appearance.actor.fame 和 appearance.movie.quality，否则停止提议。

UntypedScheme 提议方案中如何继续的决定不依赖于为所提议的第一个元素选择的值。

遗憾的是，我们的演员与电影示例仍然有一个技术上的障碍，这是因为 Figaro MH 算法的工作方式。我们再来看看 award 属性的定义：

```
class Appearance(val actor: Actor, val movie: Movie) {
    val award: Element[Boolean] =
      CPD(movie.quality, actor.famous,
          ('low, false) -> Flip(0.001),
          ('low, true) -> Flip(0.01),
          ('medium, false) -> Flip(0.01),
          ('medium, true) -> Flip(0.05),
          ('high, false) -> Flip(0.05),
          ('high, true) -> Flip(0.2))
}
```

问题是，在这个程序中 Appearance 的 award 属性定义为一个 CPD。CPD 是一个确定性的元素；虽然 CPD 中有一些 Flip，但是如果这些 Flip 元素的值是固定的，CPD 的值就是完全确定的。CPD 内的 Flip 是 CPD 元素的参数。提议一个元素时，参数保持固定，对元素值进行抽样。因为 CPD 是确定的，在 Flip 值给定的情况下，抽样的始终是相同值。结果是，您无法提议 award 值的切换。

上面的解释有些技术性，本质就是：**总是尝试提议非确定性元素**。除了 Constant 之外，原子元素通常是非确定性元素，而复合元素通常是确定性元素。CPD 是一个确定性的复合元素。有少数几个复合元素是非确定性的：复合 Flip、复合 Select 和复合 Dist。但是，其他复合元素如复合 Normal 是由 Chain 定义的确定性元素。

如何使 award 成为非确定性元素?窍门就是将程序转换为一个 award 不确定的程序。这可以通过如下定义实现：

```
class Appearance(val actor: Actor, val movie: Movie) {
  def getProb(quality: Symbol, famous: Boolean): Double =
    (quality, famous) match {
      case ('low, false) => 0.001
      case ('low, true) => 0.01
      case ('medium, false) => 0.01
      case ('medium, true) => 0.05
      case ('high, false) => 0.05
      case ('high, true) => 0.2
    }
  val probability: Element[Double] =
    Apply(movie.quality, actor.famous,
          (q: Symbol, f: Boolean) => getProb(q, f))
  val award: Element[Boolean] =
    Flip(probability)
}
```

根据电影质量和演员声望确定所用概率的函数

表示所使用概率的元素

以对应的概率，掷硬币选择 award 值

很容易看出，这个定义与 award 的原始定义等价。在原始定义中，CPD 的每个子句都以某种概率生成一个 Flip。在新定义中也使用 Flip，其概率等于每种情况的概率。这个 Flip 是一个复合 Flip，正如上面所讲述的，它是一个非确定性元素。所以提议这个 Flip 可能造成值的切换，使得 MH 可以很好地工作于这个例子。

11.4.2　避免硬条件

即使有了上一小节中介绍的自定义提议方案和程序变化，我们的例子仍然不能正确地工作。问题如下：假定您从 4 部电影得奖的状态开始。在任何一步中，利用我们的自定义提议方案，您所能改变的得奖情况最多为 2 个。在这一步结束时至少有两部电影得奖，提议也不能满足唯一获奖者的条件，将被自动拒绝。因此，您永远无法转移到满足该条件的状态。

问题的症结是，提议必须绝对满足一个硬条件才能被接受。在 11.2.3 小节中已经看到，硬条件是重要性抽样的一个问题。对 MCMC 也是如此。由于硬条件的存在，找出在第一次就满足该条件的任何状态都很难。而且，在算法找到一个这类状态之后，难以转移到其他类似的状态。

解决方案是用软约束替代硬条件。理想状况下，约束将引导算法转移到最佳的状态，在我们的例子中可以使用如下约束：

```
def uniqueAwardConstraint(awards: List[Boolean]) = {
  val n = awards.count(b => b)
  if (n == 0) 0.0 else 1.0 / (n * n)
}
```

这个约束计算 award 值为 true 的个数，并将其赋值给变量 n。然后根据 n 返回一个

得分。如果 n 非零，得分为 $1/n^2$。如果 n=1，得分为 1，得分随着 n 的增加而快速减小。同时，如果 n 为 0，则得分为 0，根据这一约束，这种状态不可能出现。

这个约束有什么效果？让我们来考虑一下起始状态。

- 如果从得奖数为 0 的状态开始，至少有一次演出得奖的任何新状态都将被接受。
- 如果从有一次演出得奖的状态开始，算法通常只接受有一次演出得奖的新状态，有时（概率为 1/4）接受有两次演出得奖的新状态，接受三次演出得奖的新状态的概率更低（1/9）。
- 如果从超过一次演出得奖的状态开始，算法始终接受得奖演出数较少的状态，因为那样约束值较高。算法偶尔也会接受具有更高获奖演出数的状态，但是通常来说其趋势是转移到只有一次演出获奖的“良好”状态。

您可以将这个约束视为“润滑剂”，有助于马尔科夫链状态间的流转，使其趋向于有一次演出获奖的“良好”状态。没有这一润滑剂，链将变得僵化，无法正常移动。将约束作为润滑剂是常用的技术。它不仅有助于从一开始就达到高概率状态，还可以在找到高概率状态之后自由地在这些状态中移动。

您可以在本书代码库中找到本例的代码。如本例所示，您有时候需要进行相当漫长的工作，包括自定义提议、程序变换、调整约束，才能使 MCMC 正常工作。如果幸运，默认的提议方案就是可行的，但是如果事与愿违，使 MCMC 算法正常工作的难度显然高于其他算法。这是我将其作为最后手段的主要原因。但是，如果您愿意投入精力，回报也很可观，您所得到的结果是其他任何算法所无法得到的。

11.4.3 MH 的应用

MH 广泛适用于各类问题，但是正如前面所讨论的，将其应用到给定的问题上可能需要花费一定的精力。每个应用都是独一无二的，遗憾的是，我不知道任何能够使特定应用运转良好的“魔法公式”。可以说，最适合于 MH 的应用场景是：观测值是变量间许多小的相互作用的结果，而不依赖于许多变量的组合。在前一小节中，您看到唯一获奖者条件依赖于获奖变量的组合，需要很大的工作量才能确定使用的正确组合。如果不必如此，MH 工作起来就容易多了。

回想 11.2.4 小节中的足球赛预测模型。在那一小节中我曾经说过，重要性抽样在唯一的观测值是联赛排名表时可以工作得很好，但是如果观察到所有比赛的结果，它就无能为力了，此时 MH 可能更好。实际上，MH 是这类问题的现成解决方案。模型中的随机变量有哪些？球员的能力和球队的其他特征以及所有比赛的结果。任何变量都没有控制所有其他变量，比赛的结果以累积方式依赖于其他变量。例如，不管其他变量的值是多少，如果球队的中锋很出色，都可能使他们赢得更多比赛。默认的提议方案可能很适合于这个问题。

图像与视频分析是 MH 的实际应用领域之一。如果您想要识别图像中的一个物体，最好使用非概率方法，如神经网络。但是如果想要更深入地解读图像或者视频（例如，识别物体或者活动之间的关系），概率模型是有好处的，可以提供用于解释图像的知识。例如，考虑球员射门、守门员扑救的动作序列。动态概率模型可以编码这种事件序列的概率。序列中的每个特定事件将产生作为证据的图像或者视频属性。概率推理算法可以根据证据推断可能的动作序列。

这类任务通常使用 MCMC 算法，原因多种多样。首先，动作序列的结构灵活多变，难以用少数随机变量捕捉，因此因子分解算法不太符合要求。其次，图像或者视频中表示亮度和颜色的变量有许多可能值，这同样是因子分解算法难以解决的。最后，重要性抽样也不适合，因为组成图像或者视频的证据概率很低。

MH 在这种问题上的实际应用可以将动作序列表示为提议目的的一个单位。例如，假定当前动作序列为中锋将球射向球门左侧，守门员向自己的右侧（中锋的左侧）鱼跃扑救。如果接着提议中锋向右射门，守门员继续向自己的右侧扑救就没有意义了。因此，您需要一个将守门员移向左侧的提议方案。精心地制定提议方案，MH 就可以有效地工作。

MH 在科学上的应用：MH 算法在生物和物理等科学上特别受欢迎。例如，在生物学中，概率模型用于捕捉生物体基因型之间的关系，基因型代表生物体的遗传物质以及表现这些基因如何在生物体中产生特质的表型。找出何种基因导致某种有趣特质，是一种有趣的查询。令这个问题复杂化的是，我们不总能知道特定生物体所具备的基因版本，但是相同染色体上相近的基因是相互关联的。不同的基因版本和特质可以用随机变量表示，从而得出基因型和表型上的一个联合概率模型。有了这个模型，就可以使用 MH 学习基因和特质之间的一般关系，推断指定个体具备的基因版本。

在前两章中，您已经学习了许多关于概率编程中推理算法的知识。我将重点放在计算给定证据下查询变量后验概率分布的问题上，但是其他查询也很有用，如找出变量的最可能值，或者计算证据的概率。这些任务各自需要不同的算法，它们都是这两章中您所看到的算法的变种。从数据中学习概率程序参数的能力也很重要，下一章将介绍如何完成上述的任务。

11.5 小结

- 抽样算法生成从可能世界的概率分布中提取的状态，观察样本中取值的频度以回答查询。
- 平均起来，样本代表的分布将等于真实分布，但是任何给定的样本分布可能与真实分布存在一定的差异。

- 生成的样本越多，可以预期抽样分布越接近于真实分布，查询的答案也就越准确。遗憾的是，获得您所需要的近似答案可能需要许多样本。
- 拒绝抽样是一种简单的抽样算法，从程序定义的先验概率分布中生成样本，并拒绝不满足条件的所有样本。它不能处理约束，不适合条件众多的情况。
- 重要性抽样使用加权样本考虑约束，尽可能避免拒绝，性能优于拒绝抽样。但是对于具有难以满足的条件或者约束值较小的情况，由于有效样本数较小，重要性抽样可能需要许多样本。
- 马尔科夫链蒙特卡洛（MCMC）算法在进行中重用其工作成果。它随机地搜索状态空间的高概率区域，找到该区域之后倾向于在其中移动。在具有难以满足的条件或者约束值较小的情况下，MCMC 的性能优于重要性抽样。
- MCMC 可能需要大量样本，难以理解和控制。在真实问题上使用 MCMC 往往需要额外的工作量，例如自定义提议和调整约束。

11.6 练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．对于如下程序：

```
val x = Flip(0.8)
val y = Flip(0.6)
val z = If(x === y, Flip(0.9), Flip(0.1))
z.observe(false)
```

a）使用变量消除法，计算给定 z 观测值的情况下，y 为真的准确后验概率。

b）使用 1000 个、2000 个…10000 个样本，运行重要性抽样。对于每种数量的样本，试验 100 次。计算给定样本数量的重要性抽样**误差均方根**，定义如下：

i. 对于每次试验，计量重要性抽样估算的概率和变量消除法计算的准确概率之间的差值，这就是误差。

ii．计算每次试验的误差的平方。

iii．计算所有误差平方的均值。

iv．取均值的平方根。这就是误差的均方根，这是计量推理算法误差的常用手段。

c）绘制均方根误差的图表。注意到什么趋势了吗？

2. 用 Metropolis-Hastings 的默认提议方案重复练习 1。这次，我们使用 10000、20000…100000 个样本。

3．现在，我们对程序略加修改，使数值参数更加极端：

```
val x = Flip(0.999)
val y = Flip(0.99)
```

```
val z = If(x === y, Flip(0.9999), Flip(0.0001))
z.observe(false)
```

a）运行变量消除法获得 y 的准确后验概率。

b）运行 100 万个样本的重要性抽样算法。

c）在这个程序中，证据的概率很低，但是您应该发现重要性抽样在此很精确。您认为这是什么原因？

4. 用默认提议方案和 1000 万个样本在同一个程序上运行 Metropolis-Hastings 算法。您应该发现结果很不好。（这可能不是每次都发生，但是在多次运行之后，结果趋向于不好）您认为为什么这个问题难以用 Metropolis-Hastings 算法解决？

5．尝试为 Metropolis-Hastings 编写一个自定义提议方案：

a）因为我们将提议 z 的定义中的 Flip 元素，所以必须使它们成为可以引用的单独变量。将 Flip(0.9999)转换成变量 z1, Flip(0.0001)转换成变量 z2，在 z 的定义中使用这些新变量。

b）创建一个自定义提议方案，表现如下：

i．提议 z1 的概率为 0.1。

ii．提议 z2 的概率为 0.1。

iii．同时提议 x 和 y 的概率为 0.8。

c）用这个提议方案运行 Metropolis-Hastings 算法，结果应该更好。您认为这是什么原因？

6．第 6 章包含了一个练习，编写概率程序以表现扫雷游戏。尝试用这个程序计算一个方块中包含地雷的概率。试验不同的算法。

第 12 章　处理其他推理任务

本章介绍如下内容：

- 如何查询多变量联合概率
- 如何计算模型中所有变量的最可能值
- 如何计算观察到的证据的概率

迄今为止，我们已经研究了回答诸如“已知打印机打印不正确的情况下，打印机电源开关关闭的概率有多大？”或者“在技艺和电影质量已知的情况下，赢得奥斯卡奖的概率有多大？”之类问题的方法。所有这些问题都可以总结为根据证据回答关于单一变量上后验概率的查询。使用概率编程还可以执行各种各样的其他推理任务，包括：

- 计算多变量上的联合概率分布。
- 根据证据计算变量的最可能值。
- 计算证据的概率（由模型生成的可能世界满足证据的概率）。
- 随时监控动态系统状态。
- 学习模型参数用于之后的推理。

本章展示执行前三种任务的方法。剩下的两种任务将在下一章中介绍。对每种任务，我描述其定义和用途，提供示例，然后说明算法的工作方式和在 Figaro 中的执行方法。幸运的是，前几章中学到的算法原理可以应用到这些任务上，所以本章中的材料对您来说应该很熟悉，并增加了一些有趣的新思路。

为了本章的学习，您应该对 Figaro 建模有基本的了解，但是通常不需要使用高级的技术。因为本章中的算法是在第 10 章和第 11 章的基础上构建的，您应该对这些章节有

很好的理解。

12.1 计算联合分布

迄今为止，您已经看到了计算单变量概率分布的各种算法。这种分布被称作该变量上的**边缘分布**。在调用各种算法（如变量消除法或者重要性抽样）时，您提供一个查询目标列表。例如，假定您有一个预测不同产品线销售量的概率程序。您将提供产品线的销售量元素，作为查询目标。然后，您可以查询给定产品线销售量超出某一水平或者符合产品线预期销售量水平的概率。

但是，如果您要查询多个变量的联合分布该怎么做？为什么您想要这么做？因为联合分布包含单个边缘分布所没有的信息。例如，您可能对两个产品线的销售量同时走低的概率感兴趣，因为这对您的业务特别不好。下面是一些联合分布的可能性。

- 两个产品线的销售量相互独立，一个产品线销售量的高低对其他产品线的销售量没有影响。
- 两个产品线的销售量相关，因此一个产品线销售量低，另一个产品线的销售量也很可能较低。这表明了一种高风险的情况。
- 两个产品线的销售量负相关，因此一个产品线的销售量低，另一个产品线的销售量很可能高。这是风险较小的情况，因为两个产品线的销售量不太可能同时走低。

仅仅知道单独的边缘分布，无法知道将会出现上述的哪一种情况。那么，如何查询联合分布呢？

Figaro 没有计算联合分布的专用算法，您必须创建一个特殊元素以捕捉想要查询的联合分布中各个元素的共同表现。然后，可以使用前几章介绍的任何一种推理算法。要创建这个特殊元素，就需要使用 Figaro 的元组构造程序——^^运算符。

例如，假定您有一个销售量元素的数组。下面的代码创建前两个销售量元素的配对：

```
val salesPair = ^^(sales(0), sales(1))
```

如下代码创建销售量元素的四元组：

```
val sales4Tuple = ^^(sales(0), sales(1), sales(2), sales(3))
```

注意：元组构造程序^^定义为最多 5 个参数。如果需要超出 5 个，可以使用嵌套元组，如 ^^(^^(sales(0), sales(1), sales(2)), ^^(sales(3), sales(4), sales(5)))。

之后，您可以将这些元组作为取值是元组的单个元素进行查询，例如：

```
VariableElimination.probability(salesPair,
  (pair: (Double, Double)) => pair._1 < 100 && pair._2 < 100)
```

对于因子分解算法，创建元组并进行有关的推理是需要付出代价的。创建元组等同

于创建一个在元组所有成分之上的因子。因此，对于 sales4Tuple，您有一个 sales(0)、sales(1)、sales(2)和 sales(3)之上的因子。您可能已经有了这样的因子，在这种情况下没有问题。但是如果没有这样的因子，创建这个因子就会显著增大算法的代价。

对于抽样算法来说，这不是问题。您只是创建元组中每个成分的选择值，成分的每个组合都被转换为元组的一个值。所以如果发现因子分解算法在计算联合概率时代价过高，可以尝试抽样算法。

一般来说，随着加入元素数量的增加，元组可能值的数量呈指数式增长。许多变量的联合分布不仅难以计算，也难以解读。如果您发现自己想要查询许多变量的联合分布，可以尝试用摘要统计代替。例如，与其计算许多产品线销售量的联合分布，为什么不计算总销售量上的分布呢？这很容易用 Figaro 集合实现。例如：

```
val allSales = Container(sales:_*)
val totalSales =
  allSales.foldLeft((x: Int, y: Int) => x + y)
println(Importance.probability(totalSales,
               (i: Int) => i > 1000))
```

:_*标记法将销售量数组转换为 Container 构造程序的可变参数列表

foldLeft 方法加总所有销售量元素，得到总销售量元素

注意：您可以在本书代码库中找到说明本节概念的完整代码示例，对于本章的所有小节都是如此。

12.2　计算最可能的解释

有时候，您想要知道的不是结果上的概率分布，而是哪一种结果最有可能发生。考虑打印机诊断问题，您观察打印机的症状，例如，打印缓慢。这种情况下，概率推理的目标是找出系统最有可能的状态（导致观察到的症状的打印机、软件、网络及用户状态）。确定最可能状态，可以告诉您所见问题的最可能根源，从而加以修复。

告诉您模型中各变量最可能状态的查询称作**最可能解释**（MPE）。当您进行诊断时，希望得到总体最佳的假设，以便知道如何修复问题。您通常运行图 12-1 中描述的如下过程。

1．观察初始症状，作为模型中的证据。例如，打印机用户可能投诉没有任何打印结果，所以在打印机网络中观察到 Print Result Summary （打印结果摘要）变量“无结果”的证据。

2．计算 MPE 确定系统的最可能状态。图中显示，在最可能的解释中，打印机电源按钮关闭，但是其他打印效果不佳的可能根源都没有出现。所以，没有打印结果的最可能原因是打印机电源关闭。

3．检查第 2 步中识别的故障是不是真正的故障，并尝试修复。在我们的例子中，用户可以检查电源按钮是否打开。

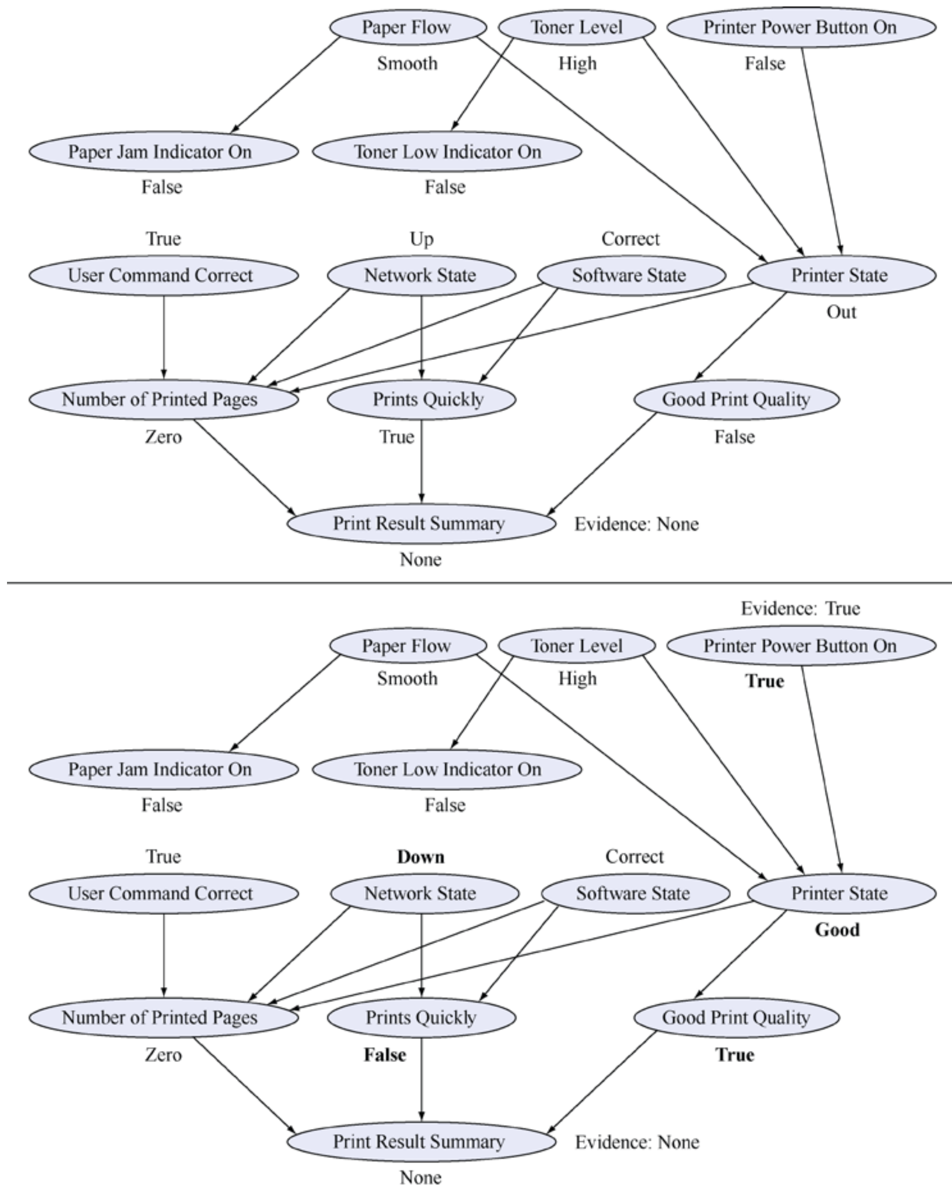


图 12-1 使用 MPE 进行诊断。上半部分展示了第 5 章中打印机网络的第一次推理。用户投诉称没有任何打印结果，所以 Print Result Summary 变量展示的证据为“无结果”。运行 MPE 查询，图中展示了每个变量的最可能值，根据 MPE，最可能的故障是打印机电源按钮关闭。执行一次测试以观察按钮是开还是关，发现电源按钮打开。所以 Printer Power Button On 变量的证据为“true”。图中下半部分显示了利用这一附加证据进行的 MPE 查询。第一次查询之后发生变化的最可能值用粗体显示。现在，最可能的故障是网络状态为“中断”

4．检查问题是否已经修复。如果修复，任务完成。在我们的例子中，电源按钮不是问题所在，所以打印机问题尚未解决。

5．如果不能解决问题，增加附加证据并返回第 2 步。插图中的下半部分显示增加的证据——Printer Power Button On（打印机电源按钮开启）变量值为 true，以及由此产生的 MPE。在这个 MPE 中，最有可能的故障是网络中断。用户可以检查该故障并尝试更正，在我们的例子中，用户发现网络电缆被拔出，将其插回之后问题解决。

再举个例子。假定您有一个降质图像，希望将其补全。可以在观察到的像素基础上，使用概率推理恢复真实图像。您有几种选择。

- 计算单个像素的边缘概率分布。尽管这在计算上是可行的，但是对于恢复整幅图像并不理想，因为只能给出每个像素的单独分布，不能告诉您哪一个像素值更有可能出现。您对某些像素的选择可能影响对其他像素的选择，这并不能从边缘分布中捕捉。
- 计算图像中所有像素的联合概率分布。遗憾的是，这在计算上不可行。所有像素的可能值数量与像素数成指数关系，可能是相当大的数字。
- 计算图像中所有像素的最可能**联合**值。这个值（MPE）是所有像素加在一起的最可能值，而不是单个像素最可能值的组合。这种方法的好处是从计算上可行，因为您只计算一个值。此外，MPE 捕捉像素之间的相互关系，这一点和单独的边缘分布不同。

表 12-1 展示了 MPE 的例子。例中有两个像素，其状态可能为 On（开）或者 Off（关）。表中展示了 2 个像素的 4 种可能世界以及相关的概率。如果只考虑像素 1，状态为 Off 的可能世界概率为 0.65，On 的可能世界概率为 0.35。所以像素 1 的最可能值是 Off。类似地，像素 2 的最可能值为 Off。但是像素 1 为 Off、像素 2 为 On 的可能世界概率为 0.45，这是所有可能世界中概率最高的，所以它是 MPE。您可以很清楚地看到，MPE 不是单个变量最可能值的组合。

表 12-1 最可能解释不一定是单个变量最可能值的组合。表中，每个像素的最可能值都是 Off，但是最可能解释是像素 1 为 Off，像素 2 为 On

像素 1	像素 2	概 率
Off	Off	0.2
Off	On	0.45
On	Off	0.35
On	On	0

总而言之，MPE 是许多应用中很有用的一类查询。我们来看看如何在 Figaro 中计算 MPE。

12.2.1　在 Figaro 中计算和查询 MPE

Figaro 中计算和查询 MPE 的接口很简单。Figaro 包含 3 个计算 MPE 的算法：两种因子分解算法 MPEVariableElimination 和 MPEBeliefPropagation，以及抽样算法 MetropolisHastingsAnnealer。我将在下一小节描述这些算法的工作方式，首先介绍其用法。

每个算法都是 MPEAlgorithm 的一个实例。和其他算法一样，MPEAlgorithm 有一次性和任意时间变种。MPEBeliefPropagation 和 MetropolisHastingsAnnealer 都有一次性和任意时间变种，但是 MPEVariableElimination 只有一次性版本。下面的例子说明了一次性算法的使用方法，但是和往常一样，可以相同的方式使用任意时间版本。

回顾第 5 章中的图像恢复场景。在这一场景中，您试图预测图像中某个像素的颜色。该图像由一个马尔科夫网络定义。每个像素有一个一元约束，规定像素为 On（开）的概率是 0.4。每对相邻像素也有一个二元约束，规定在其他条件相同的情况下，两个像素相同的概率两倍于不同的概率。这个例子的马尔科夫网络如图 12-2 所示，其中的数组从像素 11 开始，和第 5 章中的场景相同。

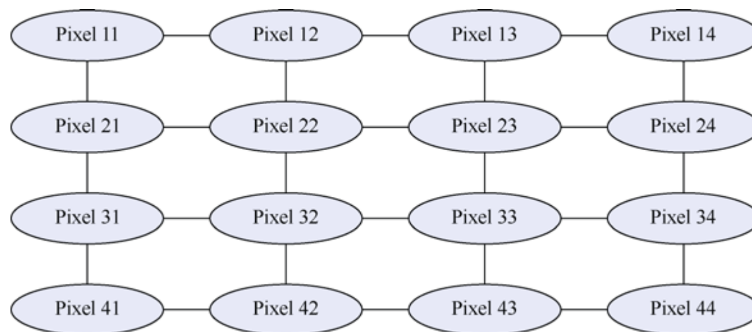


图 12-2　重现第 5 章的像素恢复马尔科夫网络

下面的程序清单展示了说明如何使用上述算法计算图像恢复示例 MPE 方法的代码。在这些清单中，像素索引从 00 开始。

程序清单 12-1　用 MPE 恢复图像

```
val pixels = Array.fill(4, 4)(Flip(0.4))

def makeConstraint(pixel1: Element[Boolean],
                   pixel2: Element[Boolean]) {
  val pairElem = ^^(pixel1, pixel2)
  pairElem.setConstraint(pair
      => if (pair._1 == pair._2) 1.0 else 0.5)
}
```

创建像素元素并为其添加一元约束

在像素对上添加二元约束

```
for {
  i <- 0 until 4
  j <- 0 until 4
} {
  if (i > 0) makeConstraint(pixels(i-1)(j), pixels(i)(j))
  if (j > 0) makeConstraint(pixels(i)(j-1), pixels(i)(j))
}

pixels(0)(0).observe(true)
pixels(0)(2).observe(false)
pixels(1)(1).observe(true)
pixels(2)(0).observe(true)
pixels(2)(3).observe(false)
pixels(3)(1).observe(true)

def run(algorithm: OneTimeMPE) {
  algorithm.start()
  for { i <- 0 until 4 } {
    for { j <- 0 until 4 } {
      print(algorithm.mostLikelyValue(pixels(i)(j)))
      print("\t")
    }
    println()
  }
  println()
  algorithm.kill()
}

def main(args: Array[String]) {
  println("MPE variable elimination")
  run(MPEVariableElimination())
  println("MPE belief propagation")
  run(MPEBeliefPropagation(10))
  println("Simulated annealing")
  run(MetropolisHastingsAnnealer(100000, ProposalScheme.default,
                Schedule.default(1.0)))
}
```

对相邻像素应用二元约束

观察一些像素的值，作为证据

这个方法对前几行中定义的模型应用给定算法。像素是模型中的元素，算法将应用到它们

创建并启动一次性 MPE 算法，计算 MPE。调用完成之后，将计算出每个像素的最可能值

查询指定像素的最可能值并打印结果

MPEVariableElimination 没有参数

MPEBeliefPropagation 以 BP 迭代次数为参数

MetropolisHastingsAnnealer 的 3 个参数将在下一小节描述

在程序清单 12-1 中可以看到，计算和查询 MPE 的模式与常规的概率查询类似。

1．创建所有元素。

2．添加作为模型定义一部分的条件和约束。

3．观察证据。

4．创建对应算法的实例。

5．启动算法。

6．查询单独元素的最可能值。虽然一次查询一个元素，但是得到的是这些元素值最可能出现的联合状态。

运行上述程序将生成如下输出：

```
MPE variable elimination
true true false false
true true false false
true true false false
true true false false

MPE belief propagation
true true false false
true true false false
true true false false
true true false false

Simulated annealing
true true false false
true true false false
true true false false
true true false false
```

这个例子特别简单，所有算法都很快地生成正确的答案。但是在更有挑战性的问题上，每种算法各有利弊。下面我们将更仔细地观察这些算法，了解它们在何时工作得更好。

12.2.2　MPE 查询算法的使用

解答 MPE 查询的算法通常是常规概率查询算法的变种。和常规查询一样，可以使用因子分解算法或者抽样算法。我将首先描述两种因子分解算法，然后描述抽样算法。因为您已经了解了这些算法的基本原理，我就不再像第 10 章和第 11 章那样详细解说了，而是指出其主要思路。

MPE 因子分解算法

MPE 的目标是计算具有最高概率的可能世界。例如，假定一个可能世界包含像素 00 的值 p_{00}，像素 01 的值 p_{01}……直到 p_{33}。这个可能世界的概率为 P(Pixel 00 = p_{00}, Pixel 01 = p_{01}, …, Pixel 33 = p_{33})。您希望最大化这个概率，这意味着您希望知道：

$$\max_{p_{00}} \max_{p_{01}} \ldots \max_{p_{33}} P(\text{Pixel } 00 = p_{00}, \text{Pixel } 01 = p_{01}, \ldots, \text{Pixel } 33 = p_{33})$$

从第 10 章中知道，可能世界的概率是因子的乘积，在这个例子中就是单个像素上一元因子以及相邻像素上二元因子的乘积。任何可能世界的最大概率就是因子乘积的最大值。这被称作**最大乘积**表达式。如果一元因子为 U，二元因子为 B，可以写出如下的最大乘积表达式：

$$U(\text{Pixel } 00 = p_{00})U(\text{Pixel } 01 = p_{01}) \ldots U(\text{Pixel } 33 = p_{33}) \times$$

$$\max_{p_{00}} \max_{p_{01}} \ldots \max_{p_{33}} B(\text{Pixel } 00 = p_{00}, \text{Pixel } 01 = p_{01}) \ldots B(\text{Pixel } 32 = p_{32}, \text{Pixel } 33 = p_{33}) \times B(\text{Pixel } 00 = p_{00}, \text{Pixel } 10 = p_{10}) \ldots B(\text{Pixel } 23 = p_{23}, \text{Pixel } 33 = p_{33})$$

在这个表达式中，第一行包含一元因子，第二行包含水平相邻的像素对上的二元因子，第三行包含垂直相邻的像素对上的二元因子。总之，您取的是所有变量上这些因子乘积的最大值。

正如常规的变量消除法（VE）是通过操纵一个乘积之和表达式运行的，MPE VE 是通过操纵最大乘积表达式运行的。它每次消去一个变量。变量消除过程如图 12-3 所示，开始的方式与常规变量消除法类似：首先将提及该变量的所有因子相乘。然后，求取消去变量的最大值而不是计算其总和。这会生成在包含被消去变量的因子中提及的所有变量上的一个因子。在本例中，创建的是像素 21 和像素 12 上的因子。

图 12-3　MPE VE 算法中消去像素 11 的结果。在像素 21 和像素 12 上创建了一个新因子。此外，创建一个回溯表，可以在像素 21 和像素 12 的值已知之后恢复像素 11 的最可能值

计算变量最大值和加总变量类似。在这两种情况下，都从因子中删除给定的变量。在加总的情况下，对于其他变量值的每个组合，该因子都包含与那些值相关的数字的总和。类似地，在求最大值的情况下，该因子包含与每个其余变量值相关数字的最大值。

表 12-2 中的例子说明了上述过程，该表中有两个变量——像素 1 和像素 2，由表中的因子联系。表 12-3 展示了求取像素 2 最大值得到的因子。这个因子中的变量取自所有提及被消去变量的因子（除了被消去变量本身）。在这个例子中，该因子中的唯一变量是像素 1。对于像素 1 的任何给定值（如 Off），因子中的条目是与该值一致的输入因子中最大的条目。与像素 1 为 Off 一致的两行中，条目分别为 0.2 和 0.45，因此最大值是 0.45。

表 12-2　消去像素 2 的输入因子

像素 1	像素 2	条　　目
Off	Off	0.2
Off	On	0.45
On	Off	0.35
On	On	0

表 12-3　从表 12-2 中的因子最大化像素 2 之后得到的因子

像素 1	条　　目
Off	0.45
On	0.35

除了计算一个变量的最大值，生成因子之外，在消去变量时还创建第二张表格。这就是**回溯表**，该表使算法可以在其他变量的最可能值确定之后，恢复被消去变量的最可能值。在解决 MPE 问题时，您不仅对找出最高概率的可能世界感兴趣，还对找出可能世界中的变量值感兴趣。回溯表的目的就在于此。对于未消去变量的每个可能值，这个列表列出了导致输入因素中最高条目的消去变量值。

在我们的例子中，回溯表如表 12-4 所示。未消去变量是像素 1，被消去变量是像素 2。该表显示对于像素 1 的每个值，导致表 12-2 中因子最大条目的像素 2 值。假定像素 1 是 Off，在输入因子中，像素 2 为 off 导致条目值为 0.2，而 On 导致条目值 0.45。所以当像素 1 为 Off，像素 2 的最大化值为 On。您在表中记录这个值。

表 12-4　从表 12-2 中的因子消去像素 2 时生成的回溯表

像素 1	像素 2
Off	On
On	Off

在过程结束时，所有变量都被消去。之后，可以通过该表回溯，找出 MPE。我们从被消去的最后一个变量开始，假定这是像素 1。因为在此变量之后没有其他变量被消去，您得到了只有一个像素 1 值的表格；假定该值为 Off。可以将该值视为像素 1 的最可能值，然后，回到前一个被消去变量，假定是像素 2。进入消去像素 2 时产生的回溯表——表 12-2 的右下角。搜索像素 1 = Off 的行，读出像素 2 的最可能值 On。以变量消除的倒序对所有变量继续上述过程，最终得到完整的 MPE。

MPE 置信传播（BP）和常规 BP 也极其相似。正如 BP，它通过在节点之间传递消息，根据消息执行因子运算工作。唯一的差别是在每个节点上执行的不是加总运算而是最大化运算。最大化运算与表 12-2 左下角所示的相同。MPE BP 往往被称作**最大乘积算法**。

关于 MPE VE 和 MPE BP，需要记住的一点是它们的属性与常规算法类似。MPE VE 是精确算法，所以如果可行，它就是最佳选择。其复杂度与常规 VE 相同：和变量消除顺序诱导图中最大团的大小成指数关系。所以它适合的问题与常规 VE 相同。对于图像恢复，最大团的大小等于图像一侧的像素数。在前面一个例子中只有 4 个像素，MPE VE 是可行的，但是随着像素数的增加，其代价将与每一侧的像素数成指数关系。

MPE BP 是一种近似算法，属性与 BP 类似。它往往工作得很好，速度也很快，但是不能保证得出最优的结果。帮助 MPE 更好工作的思路和常规 BP 相同：避免过多的环，合并元素，分解有许多父变量的 CPD，使用衰减 CPD。简化网络对 MPE VE 和 MPE BP 都有帮助。

模拟退火

最常用的抽样 MPE 算法称为模拟退火（simulated annealing，SA）算法。SA 与 Metropolis-Hastings 紧密相关。在 MH 中，抽样程序在状态空间中漫游，最终达到与该状态概率相同的一个状态。MH 找到的不一定是概率最大的状态，而是在各种高概率状态中游走。与此相反，SA 找出最可能的状态。它在状态空间中移动，通常转向概率更高的状态，但是偶尔会“倒退”到概率较低的状态。图 12-4 说明了 MH 和模拟退火运行轨迹的不同。

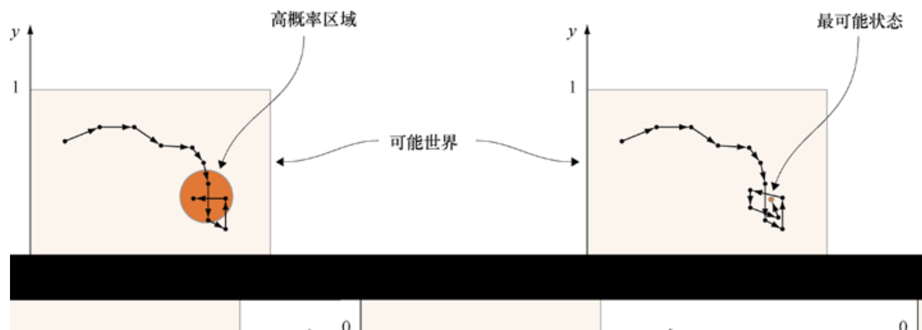


图 12-4 Metropolis-Hastings 和模拟退火算法的对比

SA 有时转移到较低概率状态的原因是为了帮助它更好地探索状态空间。如果只向高概率状态转移，可能会陷入某个局部最大值，这种状态的概率高于邻近状态，但是远离它的状态可能有高得多的概率。探索状态空间使 SA 避免陷入局部最大值，找出状态空间中总体更好的部分。SA 试图在探索状态空间和转移到高概率状态之间寻求平衡。开始时，它倾向于四处移动，最终，SA 将焦点放在找出具有高概率的邻近状态上。

SA 使用温度的概念实现上述目标。直观地说，温度控制 SA 探索状态空间的方式。如果温度很高，SA 倾向于随机探索状态空间，如果温度很低，它积极地向可能性更高

的状态转移。SA 的策略从高温度开始，随着时间推移使其冷却，所以开始时更多的是探索，最后则聚焦于“磨炼”最有可能的状态。

不同的 SA 实现在温度的使用上略有不同。下面介绍 Figaro 的实现。假定开始的概率为 p_0，提议的新状态概率为 p_1。如果温度为 t，接收新状态的概率由如下公式给出：

$$\left(\frac{p_1}{p_0}\right)^{1/t}$$

温度有何影响？我们将接收新状态的概率作为温度的函数来研究。如果 p_1 大于 p_0，这个数将大于 1，新状态总是会被接受。否则，接受概率取决于温度。如果温度 t 为无穷大，接受概率始终为 1，则始终接受新状态。这对应于纯粹的探索。如果 t=0，除非新状态和前一状态概率相同，否则新状态永远不会被接受，这对应于纯粹向高概率状态转移。在两者之间时，t 值导致探索和概率最大化的不同平衡。一般来说，t 值越小，算法探索得越少。

注意：在该算法的技术定义中，接受概率还取决于用于提议新状态的提议分布属性。在 Figaro 中，取决于提议分布属性的项通常被删去，因此，接受概率取决于状态概率之间的比例。

温度随时间变化的速度由**冷却进度表**控制。冷却进度表决定温度何时、多快下降。换言之，冷却进度表控制算法在开始最大化概率之前进行多少次探索。Figaro 提供了默认的标准冷却进度表，这足以应付大部分情况。

默认的冷却进度表有一个控制冷却速度的参数。这个参数的值越大，冷却越慢。一般来说，这个参数越大，退火花费越多的时间寻找最可能值，但是停在局部最大值的可能性越小。方法之一是从 1.0 开始，如果怀疑得到局部最大值，则增大该值。

背景：在物理学上，退火是通过逐渐降温，找出材料低能态的技术。低能态对应于高概率状态。在高温下，材料在不同能态间转移，而在较低的温度下，它试图直接降低能量。冷却进度表控制探索的不同能态数量，以及找出最小能态的速度。

您可以这样创建 SA 实例：

```
MetropolisHastingsAnnealer(100000, ProposalScheme.default,
                     Schedule.default(1.0))
```

标准 MetropolisHastingsAnnealer 构造程序有 3 个参数：样本数量、提议方案（这里我们使用默认方案）和冷却进度表（这里使用参数为 1.0 的默认进度表）。

SA 有许多和 MH 相同的属性，它是一种通用算法，可用于广泛的模型上。但是 SA 可能很慢，很难使其出色地工作。您不仅必须提供提议方案，还要指定冷却进度表。即使有了好的提议方案，冷却进度表和迭代次数之间也可能有微妙的相互作用。管理这种相互作用的最佳方式是反复尝试。

管理 SA 的技术之一是比较不同迭代次数的结果。如果一直得到相同结果，那么该结果可能就是正确的答案。如果得到不同结果，可能碰到局部最大值，应该增大参数，调慢冷却进度。如果得到相似但不完全相同的结果，SA 可能还没有收敛到最大值，应该加速冷却或者运行更多次迭代。

12.2.3 探索 MPE 算法的应用

在实践中，MPE 的计算几乎和给定证据下查询变量概率的计算一样广泛使用。应用的例子包括：

- **识别可能的行动序列**——您可能希望根据一个图像或者视频序列了解正在进行的活动。例如，如果您有一场足球赛的视频，可能想要发现其中正在进行的活动（发角球、中锋跃起头球攻门、守门员鱼跃扑救）。足球赛中可能的行动序列和结果图像可以用 HMM 表示。

  在 HMM 中，MPE 任务是根据给定的观测值，计算最可能的隐含状态序列。在第 10 章中，您已经知道 VE 因其线性结构而成为适用于 HMM 的好算法。同样，MPE VE 很适合于 HMM 中的 MPE 任务。用于 HMM 的 MPE VE 被称作 Viterbi **算法**。
- **生成自然语言句子的最可能解析**——在第 10 章中您曾经学过，PCFG 是表示句子概率模型的简单、流行框架。通过 PCFG，您可以根据模型查询句子最有可能的解析。同样，MPE VE 很适合此类问题。
- **图像分析**——您已经在本书中看到了图像修复的例子，这是 MPE 推理的典型应用之一。根据一个不完整的图像，您希望找出最可能的完整图像。例如，根据人脸的部分图像（在照片中，只能看到人脸的一部分），生成这个人的完整图像。这可能有助于识别照片中的人物。MPE BP 和 SA 都可以用于这种场景。
- **诊断**——本节开始的打印机诊断模型是诊断问题的一个例子。另一个例子是医学诊断，其目标是根据报告的症状和检查诊断患者的疾病。在诊断中，您往往希望知道所有变量的最可能状态——例如，打印机、网络、软件和用户状态的最可能组合。这有助于指明首先进行的测试和可能处理的故障。

  根据模型结构，任何 MPE 算法都可用于诊断。例如，第 10 章介绍了医学诊断的两层网络，其中疾病在第一层，症状在第二层。正如第 10 章中所述，MPE BP 是适合这种结构的出色算法。
- **科学应用**——第 11 章讨论了 MH 在生物学上的应用，其目标是根据表型（基因在人类特质上的表现）确定一个人有某种基因的概率。您可以将这种查询转换为 MPE 查询，目标是找出最有可能的基因型。正如第 11 章中的 MH 是边缘概率计算的好算法，SA 也是适合于 MAP 查询的算法。

边缘 MAP

另一类查询是给定证据下查询变量边缘概率的计算和 MAP 的组合，称作边缘 MAP。在边缘 MAP 中，您想要计算某些变量的最可能值，同时加总或者边缘化其他变量。例如，在打印机诊断应用中，您可能想要知道计算机的最可能状态，同时边缘化网络、软件和用户。

边缘 MAP 通常是很有用的查询，但是比常规边缘概率查询或者 MAP 查询更难以回答。边缘 MAP 定义涉及最大化和总计的最大乘积和表达式。在 VE 和 BP 等因子分解算法中，重要的是能够随意地移动变量。在边缘 MAP 中，问题是无法合理地在总计中移动最大化，这将严重限制算法所能执行的运算。边缘 MAP 算法是概率推理研究的重要问题。Figaro 没有包含任何边缘 MAP 算法，但是我们计划在未来加入它们。是常规活动还是入侵活动，但是遗憾的是，您没有许多攻击活动的例子以学习模型。而且，您担心新的攻击类型不同于之前见过的任何一种攻击。

12.3　计算证据的概率

除了计算查询变量上的概率分布或者最可能解释之外，有时候您还想要计算**观察到的证据的概率**。例如，假定您试图监控网络，检测攻击。您可以使用概率方法确定某个活动

在这种情况下，您可以尝试创建如下变量的联合概率分布：表示该活动是正常活动或者入侵活动以及表示活动类型的变量。但是这种模型无效，因为您不知道在确定活动为入侵式活动时，攻击的条件概率。替代方法之一是创建正常活动的模型，标记该模型下任何特别不寻常的情况。在这种“**异常检测**”方法中，您将创建正常活动的概率模型。对于某种特定活动，您将声明该活动的特性，作为模型中的证据，并询问证据的概率。如果概率足够小，可以将该活动标记为异常。这并不意味着这种活动就是入侵，而是将其标记为值得调查的事项。

证据概率计算的另一种应用是**分类**。下面我们将在 HMM 的背景下思考第 10 章中讨论的语音识别应用。在语音识别中，每个单词作为 HMM 建模，在模型中隐含状态对应于发出一个单词的各个阶段，观测值对应于生成的音频信号。为了使用贝叶斯推理确定单词的概率，需要考虑：（a）单词的先验概率，（b）单词的似然率，即给定单词观测序列的概率。似然率等于对应于单词的 HMM 中各个观测值的概率。因此，要计算单词的后验概率分布，就必须计算每个单词在 HMM 中的证据概率。

因为计算证据的概率和以前的概率推理方法有所不同，我将使用图 12-5 讲解，该图与第 1 章中的框图类似。

前两步和常规的概率推理类似：在概率模型中编码一般知识，在证据中编码关于具体情况的知识。在这个例子中，一般知识与正常网络活动相关。然后，查询是计算证据

的概率，该系统使用推理算法回答查询并返回证据的概率。概率推理的这种用法包含最后一个步骤：决定如何处理这一答案的决策模块。决策模块将编码一个策略。例如，每当证据概率低于预先确定的某个阈值时发出警报就是一个简单的策略。

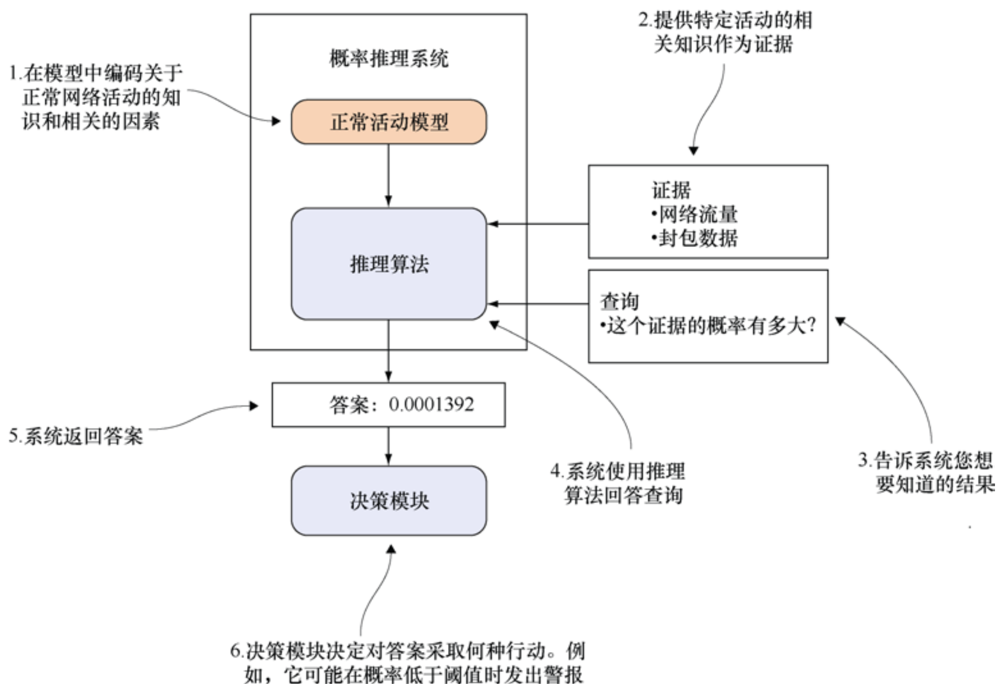


图 12-5 使用概率推理计算网络入侵检测应用中证据概率的步骤

12.3.1 观察用于证据概率计算的证据

本节继续上一节中的图像示例，但是现在您所要计算的是观察到的像素的概率。模型与之前相同。接下来，我将说明观察证据的方法。

您可能会说，您已经知道如何观察证据了——使用条件或者约束。但是条件和约束可能是模型的一部分而非证据。例如，在图像模型中，“相邻像素可能相同”的约束是模型的一部分，而不是证据。在计算证据的概率时，您不应该计算这些约束的概率，而应该计算包含这些约束的模型下的证据概率。

注意：产生这个问题的原因是，条件和约束在 Figaro 中有两种用途：修改模型和声明证据。到目前为止，这还没有成为问题，因为条件和约束的两种用途以相同的方式影响查询。但是在计算证据概率时，您必须明确地知道何为证据。

因此，Figaro 提供了一个附加的机制，以明确地指定证据。使用这一机制的步骤如下。

1．将您声明证据的元素与一个名称和一个元素集合关联。在我们的图像示例中，您将把证据应用到单个像素，所有使用如下的代码行：

```
val pixels =
  Array.tabulate(4, 4)((i: Int, j: Int) =>
    Flip(0.4)("pixel(" + i + "," + j + ")", Universe.universe))
```

这将创建一个 4×4 像素数组。每个像素为 Flip(0.4)，该元素与取决于像素索引的名称关联。例如，坐标(0, 0)的像素名为 pixel(0,0)。每个像素还与元素集合 Universe.universe 关联，该集合是默认的宇宙（记住，宇宙就是一个元素集合）。

2．将证据的每一项转换为 Evidence 类的一个实例。Evidence 类的例子包括：

—— Observation(true)，陈述值为 true 的布尔元素。

—— Condition((i: Int) => i > 0)，陈述值为正的整数元素。

—— Constraint((d: Double) => 1 / (d + 1) * (d + 1))，对双精度元素应用约束。

3．使用 NamedEvidence 类的一个实例，将证据的每一项与特定元素关联。NamedEvidence 有两个参数：对应用证据的元素的引用，以及表示证据的 Evidence 类实例。例如，如下代码指定索引（0,0）的像素值为 true 的证据：

```
NamedEvidence("pixel(0,0)", Observation(true))
```

4．将要陈述的所有 NamedEvidence 实例放到一个列表中。如下代码创建图像程序中的所有证据：

```
def makeNamedEvidence(i: Int, j: Int, obs: Boolean) =
  NamedEvidence("pixel(" + i + "," + j + ")", Observation(obs))
val evidence =
  List(makeNamedEvidence(0, 0, true),
       makeNamedEvidence(0, 2, false),
       makeNamedEvidence(1, 1, true),
       makeNamedEvidence(2, 0, true),
       makeNamedEvidence(2, 3, false),
       makeNamedEvidence(3, 1, true))
```

这样使用命名证据和使用条件及约束指定证据的作用相同。和往常一样，所有元素（本例中是像素）根据生成模型生成。然后，任何有命名证据的元素都应用一个条件或者约束。添加命名证据的效果与添加条件或者约束相同。命名证据使您可以访问应用条件或者约束的元素。

现在，您已经知道如何用命名证据的形式表达关于特定情况的知识，下面我们来看看如何提出证据概率查询和接收答案。

12.3.2　运行证据概率算法

现在，您已经为计算证据概率做好了准备。实现这一步的基本接口很简单。Figaro

提供两种计算证据概率的算法：类似于重要性抽样的抽样算法 ProbEvidenceSampler 和因子分解算法 ProbEvidenceBeliefPropagation。例如，调用如下语句可以用给定证据上的 10000 个样本运行 ProbEvidenceSampler 算法并打印结果概率：

```
println(ProbEvidenceSampler.computeProbEvidence(10000, evidence))
```

使用如下语句可以在固定的时间内（如 1 秒）运行 ProbEvidenceSampler：

```
println(ProbEvidenceSampler.computeProbEvidence(1000L, evidence))
```

1000L 中的 L 不能遗漏，这告诉 Scala 该值是一个长整数，说明它表示的是一个以毫秒为单位的时间而不是样本数。使用常规整数在固定样本数上运行算法；使用长整数则在固定时间内运行。

ProbEvidenceBeliefPropagation 与此类似，您可以简单地提供迭代次数和证据：

```
println(ProbEvidenceBeliefPropagation.computeProbEvidence(20, evidence))
```

BP 证据概率算法的优缺点和常规 BP 一样，它可能很快，但它是一个近似算法，收敛到的答案可能与正确答案有明显的差距，特别是在多环的网络中。图像网络就是如此；如果运行本书代码库中的程序，就会看到 BP 在不同抽样上给出不同的答案，即使用很大的样本数多次抽样也是如此。所以，对于本例，BP 不是一个好的算法。

抽样证据概率算法有多好？这取决于您对“好”的定义。我们对绝对误差和相对误差做如下定义。

- **绝对误差**是估算值和真值之间的差值。例如，如果真实证据概率为 0.0001，计算出的答案为 0.001，绝对误差是 0.0009，这个误差似乎不大。
- **相对误差**等于绝对误差除以真值。在这个例子中，计算出的答案（0.001）10 倍于真实概率（0.0001），所以从相对的角度看，结果不太好。确实，相对误差为 0.0009/0.0001 = 9。

用抽样法求取证据概率往往产生很好的绝对误差，但是相对误差不佳。另一种陈述就是，它可能告诉您概率接近于 0，但是难以确定小数点之后的第一个有效数字之前有几个零。抽样法的结果是否足够好取决于您的应用。对于异常检测来说，取决于正常情况的证据概率是否也接近 0。如果正常情况的概率与 0 有一定距离，抽样算法就工作得很好，因为它在某种情况不属于该类别时能够快速发现。但是如果正常情况的概率也接近于 0，异常情况更接近一些，抽样算法就遇到麻烦了。尽管这是抽样算法的一般属性，但是对证据概率计算来说问题特别严重，因为在这种计算中您往往关注的是小概率事件。

对常规概率查询中的抽样算法有帮助的技术在此也有效。最重要的是**避免硬条件**。如果模型有许多硬条件而导致满足所有条件的状态很少，抽样程序就很难找到这些条件。估算的概率通常为 0 而不是小的正数，按照绝对的说法，这是正确的，但是按照相

对的说法，这就是严重的错误。

12.4　小结

- 您可以创建一个变量元组，使用常规推理计算多个变量的联合概率分布。但是如果想要查询许多变量上的联合分布，创建一个用单独统计量总结它们的变量更好。
- 可以使用 MPE 查询计算所有变量的最可能值。Figaro 为此提供了分解和抽样（模拟退火）算法。在许多应用中，MPE 查询可以作为边缘概率查询的替代方法。
- 计算证据的概率对于异常检测和分类等应用可能有益。Figaro 也为此类任务提供了分解和抽样算法。

12.5　练习

在 www.manning.com/books/practical-probabilistic-programming 上可以找到部分练习的解答。

1．使用第 5 章的打印机诊断程序，计算已知打印结果不佳的情况下，打印机状态和网络状态的联合概率。这些变量是相关、负相关还是独立？

2．使用同一个程序，经历 12.2 小节开始时描述的诊断步骤。

3．使用同一个程序，计算打印结果不佳这一证据的概率。

4．遵照练习 10.7，用 Figaro 创建 HMM 的表现形式。根据特定的观测序列，计算生成那些观测值的隐含状态的最可能序列。

5．这个问题涉及第 5 章中的图像恢复网络。对于如下任务，试验计算证据概率的不同算法。

a）计算左上角的像素点亮的证据概率。

b）计算代码中 data 字段表示的证据概率。

第 13 章　动态推理和参数学习

本章介绍如下内容：

- 如何监控动态系统的状态
- 如何学习概率程序的参数

前一章描述了可向概率模型提出的几类不同查询。本章通过另外两种重要且广泛使用的推理任务，继续讲解这些查询。第一种任务是从传感器接收信息，随时监控动态系统的状态，在 13.1 小节中讨论。第二种任务是从数据中学习概率模型参数，在 13.2 小节中介绍。尽管用于这两类推理的技术各不相同，但是都是许多应用中所必要的高级方法。花时间了解这些方法及其使用方式是很有价值的。本章最后简单总结本书中所学的知识，并介绍 Figaro 中超越这些知识的使用方法。

13.1 小节中介绍的动态系统监控在很大程度上依赖于介绍动态概率模型的第 8 章。这一节中描述的粒子过滤算法基于第 11 章中讨论的重要性抽样。关于模型参数学习的 13.2 小节使用了第 9 章中的素材。特别是，您应该理解不同的学习方法，如最大化后验（MAP）和贝叶斯方法。

13.1　监控动态系统的状态

第 8 章讨论如何表示动态概率模型。进行有关动态模型的推理有两种方法，如图 13-1 所示。

- 如图上半部分所示，您可以在固定的时间步中展开动态模型。例如，从描述足球赛开始时的初始状态入手，可以将其展开为 90 个时间步，每一步表示 1 分钟。这就生成了一个常规概率模型，可以用前几章描述的常规手段进行推理。这种方法的主要好处是，您可以进行前向和后向推理。例如，可以使用比赛开始时的条件预测结果，或者使用比赛结果推断初始条件。
- 图中下半部分显示，您可以随时跟踪系统状态。在这种方法中，您不将模型展开到许多个时间步上，而是在每个时点维护当前和之前的时间步。当您随着时间推进时，生成新的时间步并将其作为当前时间步。原来的**当前**时间步变成**前一**时间步，原来的前一时间步则被抛弃。这种方法的主要好处是任何时候内存中仅保持 2 个时间步，可以持续任意长的时间。缺点是，您只能向前推理。

本节介绍第二种方法的使用，此类推理有好几种名称：**系统状态监控**、**状态估算**或者更具技术性的**过滤**。

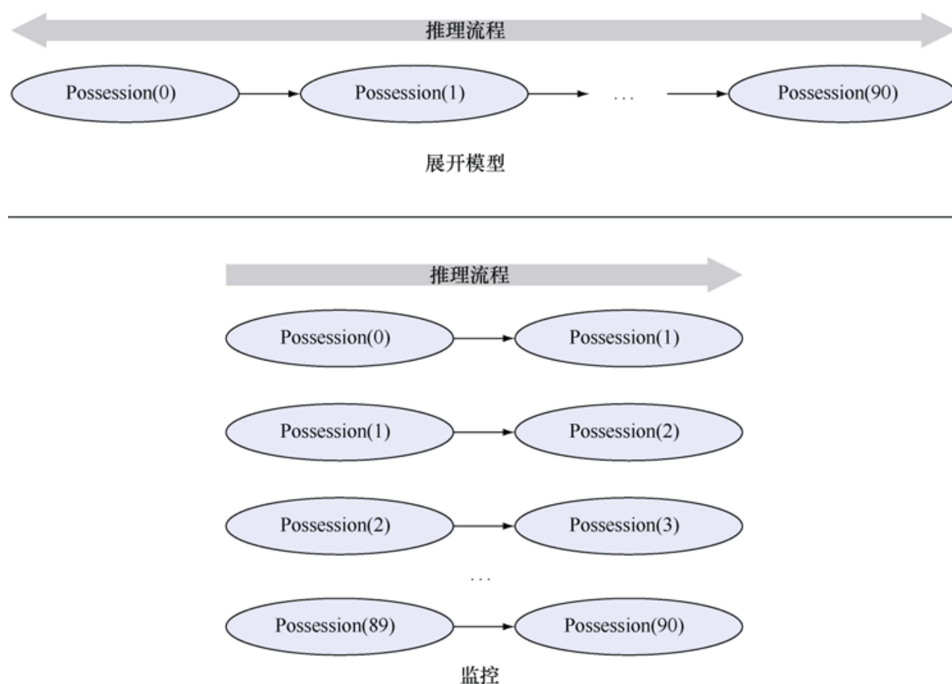


图 13-1　动态模型推理的两种方法。图中上半部分展示了将模型展开为全部时间步，创建单一贝叶斯网络的方法。灰色的箭头表示推理的流程，在这种情况下可以进行前向和后向推理。图中下半部分展示了监控活动——推理程序反复地从一个时间步转移到下一步，推理流程是单向的

13.1.1 监控机制

第 8 章介绍了执行此类推理的原理，我们在此复习一下。Figaro 中的监控使用了宇宙的概念。每个时间步都有一个不同的宇宙，往前推进时，生成新的宇宙以表示当前时间步。倒数第二个宇宙被抛弃并进行垃圾收集。为了实现这一过程，创建一个初始宇宙以表示初始状态，创建一个迁移函数，以前一个宇宙为参数返回下一个宇宙。从一个宇宙转入下一个宇宙的元素必须命名。观察或者查询的元素也必须命名。

下面是更详细的步骤，以及第 8 章中监控饭店容量的程序摘要。这个模型包含两个状态变量，它们都有名称。

- seated——整数列表，表示每组就坐客人在座位上已经停留的时间。
- waiting——整数，表示等位客人的数量。

下面是具体的步骤。

1．创建一个宇宙表示系统初始状态。为每个相关元素命名并将其放入初始宇宙。

创建表示初始状态的宇宙

```
val initial = Universe.createNew()
Constant(List(0, 5, 15, 15, 25, 30, 40, 60, 65, 75))("seated", initial)
Constant(1)("waiting", initial)
```

创建两个元素，以名称“seated”和“waiting”将其放在初始宇宙中

2．创建一个迁移函数。前一个宇宙中影响当前宇宙的任何元素都按名访问。在下一宇宙中对应的元素必须采用相同的名称。

迁移函数以前一宇宙为参数，返回下一宇宙

创建下一宇宙

```
def nextUniverse(previous: Universe): Universe = {
  val next = Universe.createNew()

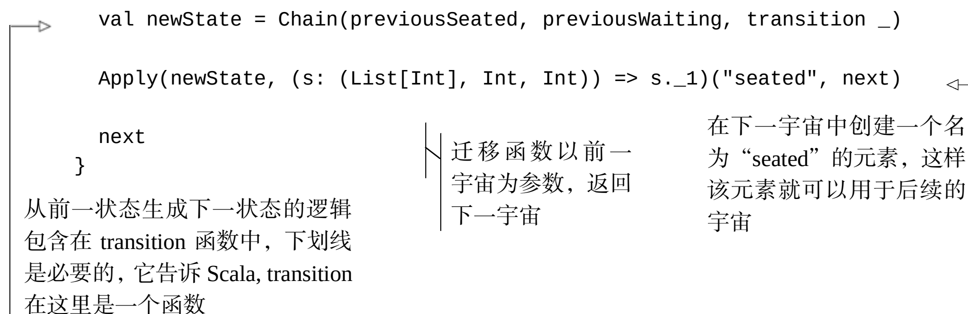
  val previousSeated = previous.get[List[Int]]("seated")
  val previousWaiting = previous.get[Int]("waiting")
```

从前一宇宙得到之前的“seated”和“waiting”元素

```
    val newState = Chain(previousSeated, previousWaiting, transition _)

    Apply(newState, (s: (List[Int], Int, Int)) => s._1)("seated", next)

    next
  }
```



3．创建监控算法的一个实例。Figaro 提供两种监控函数：粒子过滤和因子分解边界（factored frontier）。粒子过滤是一种抽样算法，而因子分解边界是因子分解算法（诚如其名）。粒子过滤是最常用的算法，我将在后面加以说明。

```
val alg = ParticleFilter(initial, nextUniverse, 10000)
```

4．启动算法，生成初始时间步上的概率分布。

```
alg.start()
```

5．用 advanceTime 方法反复推进算法，每次生成一个时间步。该方法可以在每个时间步陈述证据。证据用 NamedEvidence 接口陈述。这就是作为证据观察的元素必须命名的原因。

```
alg.advanceTime(List(NamedEvidence("waiting", Observation(1))))
```

6．在每个时间步，可以查询当前时点各个元素的分布。查询元素必须命名。例如，要得到在饭店就坐的客人列表预期长度，可以使用如下语句：

```
alg.currentExpectation("seated", (l: List[Int]) => l.length)
```

现在您已经理解了使用过滤算法的机制，下面我们来了解一下最广泛使用的过滤算法——粒子过滤。

13.1.2 粒子过滤算法

粒子过滤是基于重要性抽样的算法。**粒子**是样本的**同义词**，**过滤**则是**监控**的另一种说法，所以这是一种使用样本的监控算法。

算法的运行

粒子过滤算法的运行如图 13-2 所示。这个算法的主要概念是，任何时间点的状态分布由一组样本表示。在图中的左侧是一组表示前一状态分布的粒子，右侧是一组表示

当前状态分布的样本。两者之间有两个中间阶段。

算法的第一步是取得前一个状态的每个样本，并通过迁移函数所提供的系统动态性传播。这创建了系统当前状态的新分布，但是没有考虑当前状态所提供的证据。下一步是根据证据进行调节。这和重要性抽样的实现方式相同，但是为每个样本指定一个对应于该样本证据概率的权重。（具体地说，如果样本违反任何条件，则权重为 0；否则，权重为元素上定义的约束值的乘积）

此时，您有一组加权样本，如图 13-2 中的第 3 步所示。在图中，黑色圆圈的大小直观地说明了样本的权重。但是，要结束本算法，您还需要获得一组不加权的样本，以表示当前时间步的分布。

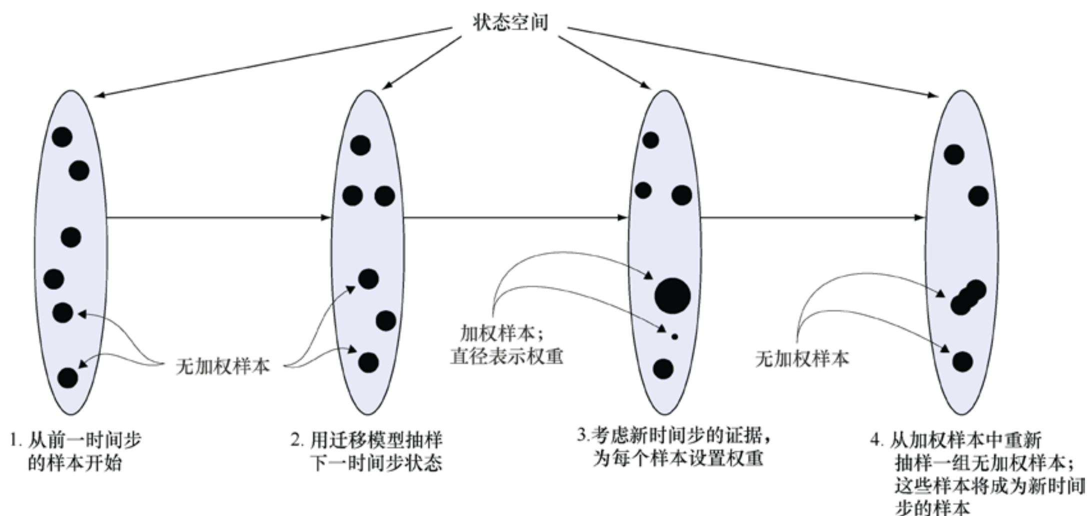


图 13-2　粒子过滤算法：本图显示了从前一时间点的状态分布估算转移到当前时间点状态分布的过程

这一步由**重新抽样**过程实现。在重新抽样中，创建一组近似于加权样本概率分布的无加权样本。该算法从一组加权样本中选择每个无加权样本。指定加权样本被选中的概率与其权重成正比。例如，假定某个加权样本的权重为 1/3，另一个样本的权重为 2/3。权重为 2/3 的样本在新的无加权样本集中出现的概率大约 2 倍于另一个样本。所以，该状态在无加权粒子中出现的概率也 2 倍于另一个状态，这和加权粒子的情况相同。

粒子过滤的属性

重新抽样是粒子过滤算法的关键组成部分。您可能觉得奇怪，为什么需要这一步？为什么不自始至终都使用加权粒子？不能通过动态性传播加权粒子，然后计算新权重，根据证据调节吗？这样做的问题是，您最终将得到一组权重极小的样本。在每个时间步，算法将把当前样本权重乘以约束值，得到更小的权重。随着时间的推移，如果过

程足够长，任何样本都不可能表现出该概率的轨迹。所以这组样本不能代表真实的状态分布。

重新抽样在每次迭代中抛弃最低概率的粒子，仅保留对真实状态更有代表性的粒子，解决了这个问题。这不能保证为您提供高概率的样本，仍然有可能选中低概率的样本，只是概率较小。但是平均起来，重新抽样过程保持高权重样本的可能性最大。所以，保持的样本通常能够表示更有可能出现的轨迹。

虽然重新抽样在这方面有益处，但是它也有降低每次迭代中样本多样性的有害效应。如果一个加权样本的权重远大于所有其他样本，下一时间步中几乎所有无加权样本都对应于该样本。如果这个样本就是系统的真实状态当然很好，但是如果不是该怎么办？粒子过滤很难从之前犯的“错误”中恢复。

这种现象称作**粒子饥饿**。下面是一个例子。假定您使用粒子过滤监控 A 队和 B 队的足球比赛状态。我们假设有一个变量表示球队的相对实力。如果 A 队首先进球，可能所有样本都表明 A 队强于 B 队。此时，如果这是错误的结论，即使 B 队射进 10 个球，算法也无法恢复！没有任何 B 队强于 A 队的样本能够留下来，图 13-3 说明在一个时间步就可能发生的粒子饥饿。

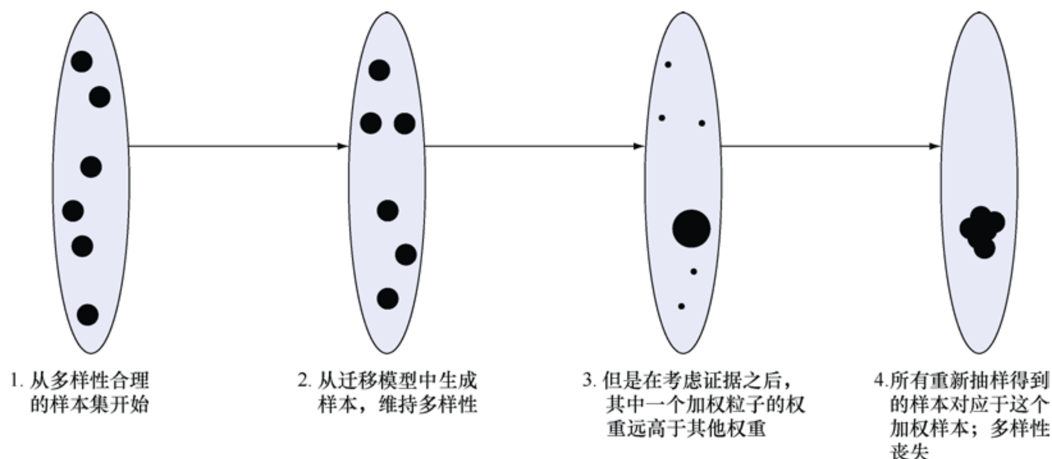


图 13-3　粒子饥饿发生在重新抽样导致样本集中的多样性丧失的时候，这可能发生在一个时间步中

正如本例所述，粒子饥饿问题对于不随时间变化的变量来说尤其严重。在刚才描述的模型中，球队的实力是固定的，所以在粒子过滤算法做出关于它的决定之后，就不会更改。这也暗示了对该问题的潜在缓解手段：引入这些变量随时间推移而变化的概率。例如，如果球队的相对实力在每分钟里有 1%的概率出现变化，即使算法在 A 队得分之后决定所有的样本都显示其实力更强，在下一时间步中某些样本也可能会出现变化。之后，当 B 队得分时，那些特殊的粒子将得到更高的权重，从而更多次地被重新抽样。最

终，如果 B 队连续得分，B 队更强的样本将会占据统治地位。

除了粒子饥饿问题之外，重要性抽样的一般考虑因素也适用于粒子过滤。如果单一时间步中的证据可能性较小，样本集就不能很好地代表后验分布。标准的缓解措施适用于这种情况。

- 避免硬条件，尽可能使用软约束。例如，观察正好有 5 个人在饭店等位可能难以满足，导致许多样本的权重为 0 而成为无效样本。做为替代，可以添加一个约束，其值在正好有 5 位客人等位时最大，并随着等位人数与 5 的距离增大而逐渐减小。
- 避免极小的约束值，弱化软约束。

这两种缓解措施的原理都相同。用更为“平滑”的近似模型代替概率模型，尽管新模型不准确，但是更容易使用抽样算法推理，结果也可能更好。当然，如果调整过度，该模型最终会远离真实模型，从而使结果无效。这种平衡行动最好通过反复尝试加以控制。

13.1.3 过滤的应用

动态模型在概率推理中十分普遍，过滤可能是最广泛使用的动态概率模型推理方法。因此，过滤存在许多可能的应用，下面是一些例子。

- **监控人或系统的健康状态**——例如，急诊室里可能有一位患者，连接到各种传感器（如温度和心率传感器）。您希望监控患者的身体状况，以便在出现问题时快速干预。患者具备隐含的变量，如失血量，监控目标是随时根据传感器读数推算这些变量。
- **机器人定位**——如果机器人正在某个环境中行进，必须根据其传感器（例如，可能包括雷达）跟踪其位置。这种应用中的隐含变量与机器人的位置和速度相关。机器人定位的目标是根据传感器维护这些隐含变量。这种应用的变种之一称作即时定位与地图构建（simultaneous localization and mapping，SLAM）。在 SLAM 中，将机器人放入一个未知环境，同时建立该环境的地图，确定机器人在环境中的位置。
- **监视和跟踪**——在监视应用中，一定数量的传感器覆盖某个区域，在区域中出现物体时接收到信号。例如，传感器可能是视频摄像头，信号可能是人或车辆的图像序列。这个应用中的隐含变量是出现在该区域的物体。根据传感器信号，您希望算出区域中有哪些物体。在相关的跟踪应用中，目标是取得物体的单独观测值，并将其转换为物体在区域中移动的轨迹记录。
- **建立复杂持续过程的模型**——过滤在建立长期运行的复杂过程模型上很有用。选举就是一个例子，在选举模型中，隐含状态包括附属于候选人的变量，如候选人在不同群体的受欢迎程度。这种隐含状态随着时间动态变化，您可以使用

建立这种系统的动态模型，每个时间步对应一天。传感器就是民意测验，在选举期间，民意调查的结果能够呈现出这种隐含状态。如果希望根据民意调查预测选举结果，可以跟踪每天的隐含变量状态。假定距离选举还有 30 天。使用过滤，就可以得出这个时间点候选人状态的概率分布。现在，您可以从当前初始状态的估算开始，将动态模型展开到 30 步，预测选举的最终结果。

13.2 学习模型参数

本书的第 3 部分以推理为焦点：根据证据计算查询的答案。概率编程系统的另一项重要任务是学习：根据数据改进模型。在学习应用中，您不知道模型中所使用的数值参数，所以使用过去的数据帮助估算参数。在贝叶斯学习范式中，使用过去的数据创建参数值的后验概率分布，而在最大似然（ML）和最大后验（MAP）范式中，使用学习算法估算用于模型的一组参数值。

我将重点放在推理而非学习上有两个原因。

- 在贝叶斯学习范式中，学习是由某个推理算法执行的。
- 在 ML 和 MAP 学习范式中，推理算法被用作学习的内循环，学习算法是围绕它的相对简单的包装器。

本节详细介绍这些要点，说明在贝叶斯和 MAP 范式中学习的方式。我将描述所使用的两种算法以及在 Figaro 中完成这些任务的方法，重点是学习模型参数的问题，对结构的学习只做简单的补充说明。

13.2.1 贝叶斯学习

在第 9 章中第一次提到的贝叶斯学习方法中，概率程序由两个主要部分组成。

- 模型参数的先验概率，我们将其称作 P_0(Parameters)。0 表示这是看到任何数据之前的参数值。
- 每个数据实例在数据实例中数据变量值上的条件概率分布。我们将其称作 P(Data | Parameters)。注意，因为这是一个概率程序，不同数据实例中的数据可能有不同的结构。例如，给定实例中的数据可能是英语句子，不同实例中的句子有不同的长度。

有了这两个部分，就可以使用链式法则定义参数和数据上的联合分布：

$$P(\text{Parameters, Data}) = P_0(\text{Parameters})\ P(\text{Data} \mid \text{Parameters})$$

然后，可以用全概率公式加总参数，得出数据上的概率分布。得到的公式如图 13-4 所示。

给定*p*的情况下，数据的概率

*p*的先验概率

$$P(\text{Data}=d)=\sum_p P_0(\text{Parameters}=p)\,P(\text{Data}=d\mid \text{Parameters}=p)$$

对参数值p求和

图 13-4　使用参数上的概率分布计算出的数据概率

注意：因为参数通常是连续变量，可能值有无穷多个，所以使用积分而非总和。在连续变量上，积分与总和类似。

数据由一组数据实例组成，每个实例由数据变量的一组值组成。给定数据的一组参数值的**似然率**是给定参数值下该数据的概率。这等于给定参数值下，所有实例中该数据概率的乘积。

贝叶斯学习中的关键概念是贝叶斯法则。贝叶斯法则指出，在数据给定的情况下，参数值 *p* 的后验概率与 *p* 的先验概率和 *p* 的似然率的乘积成正比。我们将后验概率称为 P_1(Parameters)，观察到的数据为 *d*。则后验概率可以使用图 13-5 中的公式计算。

后验　先验　似然率

$$P_1(\text{Parameters}=p)\propto P_0(\text{Parameters}=p)\,P(\text{Data}=d\mid \text{Parameters}=p)$$

成正比

图 13-5　学习参数所用的贝叶斯法则

使用贝叶斯学习预测未来的观测值

在贝叶斯方法中，计算后验分布并用它预测没有学到的未来数据实例。另一种表达方法是：对于未来的数据实例，这个后验分布将成为先验分布。我们将这个后验分布称作 P_1(Parameters)，并用 Data_1 表示未来的数据实例。使用图 13-4 所示的推理，可以得到图 13-6 中的公式。

现在，将这个公式代入图 13-5 中 P_1 的公式，可以得到从原始数据（Data_0）中学习之后的新数据（Data_1）最终预测公式，如图 13-7 所示。

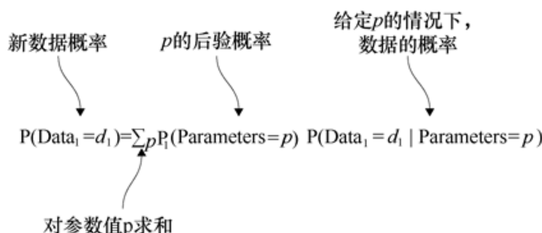


图 13-6　为了得到新数据的概率，使用参数值上的后验概率分布

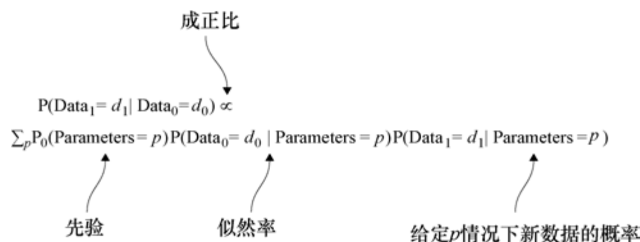


图 13-7　给定观测数据，新数据概率的最终公式

$P(Data_1 = d_1 \mid Data_0 = d_0)$称作后验预测，因为它是基于参数值后验分布做出的新数据预测。现在，重点是：**后验预测公式是一个乘积之和表达式，您可以使用任何推理算法计算。贝叶斯学习是由推理执行的，不需要某种特殊算法。**

在 Figaro 中使用贝叶斯学习

您已经知道贝叶斯学习不需要特殊算法，也已经拥有了需要的所有工具。为了说明其工作原理，我将使用第 3 章中的垃圾邮件过滤模型的简化版本。这一过程有以下几个步骤。

1．定义参数及其先验分布。

```
val spamProbability = Beta(2,3)

val wordGivenSpamProbabilities =
  featureWords.map(word => (word, Beta(2,2))).toMap
val wordGivenNormalProbabilities =
  featureWords.map(word => (word, Beta(2,2))).toMap
```

表示给定电子邮件是垃圾邮件概率的参数

表示已知电子邮件是否垃圾邮件的情况下单词出现在该电子邮件中概率的参数。featureWords 表示从训练电子邮件中得出、作为特征的单词

2．创建定义单独数据实例（即电子邮件）和给定参数下实例概率分布的类：

```
class EmailModel {
  val isSpam = Flip(spamProbability)


  val hasWordElements = {
    for { word <- featureWords } yield {
      val givenSpamProbability =
        wordGivenSpamProbabilities(word)
      val givenNormalProbability =
        wordGivenNormalProbabilities(word)
      val hasWord =
        If(isSpam,
           Flip(givenSpamProbability),
           Flip(givenNormalProbability))
        (word, hasWord)
    }
  }

  val hasWord = hasWordElements.toMap
}
```

表示单独电子邮件的类

表示电子邮件是否垃圾邮件的元素，使用垃圾邮件概率参数

根据电子邮件是否垃圾邮件，表示单词存在的元素，使用了给定垃圾或者正常邮件情况下单词存在的对应概率

创建从单词到关联元素的映射

3．创建该类的实例，每个对应于一个数据实例（训练集中的单独电子邮件）。观察电子邮件模型中每个电子邮件相关的证据。注意，训练集中的所有电子邮件使用的是相同参数的相同电子邮件模型。

```
for { email <- trainingEmails } {
  val model = new EmailModel
  for { word <- featureWords } {
    model.hasWord(word).observe(email.text.contains(word))
  }
  model.isSpam.observe(email.label == "spam")
}
```

遍历训练集中的所有电子邮件

为每个电子邮件创建一个新的电子邮件模型

观察模型中关于电子邮件文件的证据

4．为想要预测的所有未来数据实例（未来的电子邮件）创建该类的一个实例。在本例中，预测的是一个未来的邮件，但是可以扩展到任意数量。同样，这个未来的电子邮件使用和训练电子邮件相同的模型和相同的参数。

```
val futureModel = new EmailModel(dictionary)
```

5．现在，您可以按照通常的方法回答未来实例的查询。例如，您可能想要观察未来电子邮件中的单词，查询它是不是垃圾邮件。

```
for { word <- featureWords } {
  futureModel.hasWord(word).observe(futureEmail.text.contains(word))
}
println(MetropolisHastings.probability(futureModel.isSpam, true))
```

根据单词，使用学习到的参数推断未来的电子邮件是不是垃圾邮件

观察未来电子邮件中的单词，但是不观察其标签，因为那是未知的

应该使用哪一种算法？在这个例子中，我使用的是 Metropolis-Hastings 算法，它通常是贝叶斯学习的出色候选。虽然可以使用因子分解算法，但是它们将从先验分布中抽取一组参数值，将其作为唯一可能的参数值。这样做可能出现无法确定有高后验概率的参数值的风险。重要性抽样则可能因为数据实例很多、证据概率极低而难以实施。因此，Metropolis-Hastings 就成为了最佳的候选。

正如前一章所讨论的，Metropolis-Hastings 可能需要投入精力才能有效运行。本章代码库中的数据集很简单，这与第 3 章不同。对于这种简单的数据集，Metropolis-Hastings 的默认设置工作得很不错。但对于现实中的数据集，您需要使用更多样本，或者长时间运行算法。遗憾的是，难以预先知道应该运行多长时间，所以您应该尝试，发现多少样本能够给出较为一致的结果。您还可能需要使用自定义提议。

13.2.2 最大似然和 MAP 学习

本小节说明如何执行 MAP 学习，这种学习方法在第 9 章中第一次介绍。我首先将描述在 Figaro 中实施该方法的机制，然后解释所使用的算法。记住，在 MAP 学习中，您选择的是最大化先验概率与似然率乘积的参数值。然后，对未来的数据实例使用这些参数值。

注意：最大似然（ML）学习就是使用对每个值指定相同概率的均匀先验分布的 MAP 学习。所以，我对 MAP 学习所做的说明也适用于 ML 学习。

Figaro 中的 MAP 学习机制

Figaro 提供了 MAP 学习的常见模式。如图 13-8 上半部分所示，MAP 学习编码中的主要难点是将同一个模型用于学习参数和后续测试案例的推理，但是这两个模型使用不同的参数。在学习期间，参数是可学习的。例如，您可能使用一个β分布表示垃圾邮件概率。但是，在后续的推理中，参数是特定的固定值。例如，您可能学习到垃圾邮件的概率是 0.4，希望在以后使用这个值。因为参数是不同的元素，您必须创建两个不同模型，通过剪切-粘贴共享代码，这很容易出错。

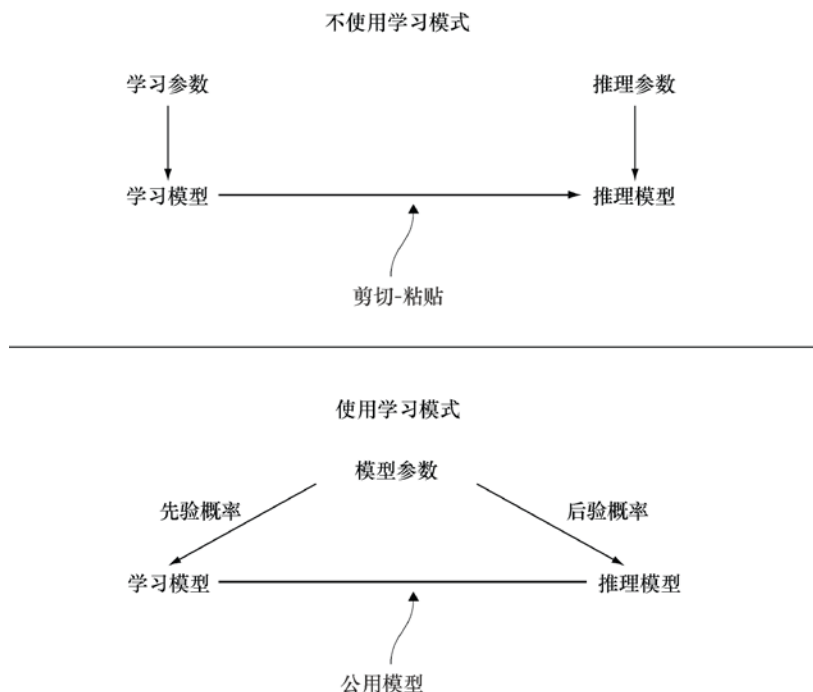


图 13-8 为学习和推理创建模型。上半部分说明没有学习模式时所应该进行的工作。因为学习和推理使用不同的参数元素，您必须创建两个模型，通过剪切-粘贴共享代码。而使用学习模式（下），参数封装在 ModelParameters 类中，使用的是共同的模型，该模型根据选择的是先验或者后验参数而专门化为学习或者推理模型

Figaro 的学习模式（如图 13-8 下半部分所示）使您可以对学习和推理使用相同模型。它使用 ParameterCollection 数据结构，这个结构是一个专门设计用于容纳模型参数的元素集合。要使用这个模式，您必须创建 ModelParameters 数据结构的一个实例。从 ModelParameters，您可以得到用于学习的 priorParameters 或者用于推理的 posteriorParameters；这两者都是一个 ParameterCollection。ParameterCollection 中的参数得到一个名称，您可以在模型中按名引用它们。

下面是您将使用的步骤。您可以在代码库的 MapLearning.scala 中看到完整的程序。

1. 创建 ModelParameters 实例。

```
val params = ModelParameters()
```

2．为每个参数命名并将其与您的 ModelParameters 关联。

```
val spamProbability = Beta(2,3)("spam probability", params)
val wordGivenSpamProbabilities =
  featureWords.map(word =>
    (word, Beta(2,2)(word + " given spam", params))).toMap
val wordGivenNormalProbabilities =
  featureWords.map(word =>
    (word, Beta(2,2)(word + " given normal", params))).toMap
```

对于每个特征词，建立两个元素，并为其命名，对应于正常和垃圾邮件。将这些元素放入两个映射中

3．让您的模型以 ParameterCollection 作为参数。

```
class EmailModel(paramCollection: ParameterCollection) {
```

4．在模型中，按名称取得参数。

```
val isSpam = Flip(paramCollection.get("spam probability"))
```

5．对于训练实例，让其使用 ModelParameters 的 priorParameters 创建模型。

```
for { email <- trainingEmails } {
  val model = new EmailModel(params.priorParameters)
```

6．为了学习 MAP 参数值，使用期望最大化（EM）算法的一个实例。正如下面所要描述的，EM 是一种“元”算法，围绕一系列推理算法中的任何一个。Figaro 包含使用 VE、BP、重要性抽样和 MH 的 EM 变种。

EM 是一种迭代算法，任何 EM 变种的第一个参数都是迭代数量。其他参数就是该算法变种通常使用的参数。例如，对于使用 BP 的 EM，您将指定 BP 迭代次数，而对使用重要性抽样的 EM，您将提供样本数量。EM 变种的最后一个参数是 ModelParameters，包含想要学习的参数。在本例中，我使用 EM 的 VE 变种。下面是创建和运行该算法的方法：

```
val learningAlg = EMWithVE(10, params)
learningAlg.start()
```

7．现在，已经学习了 MAP 参数值。您可以在用于未来数据实例的模型中使用这些后验参数值。

```
val futureModel = new EmailModel(params.posteriorParameters)
```

8．现在，您可以通常的方式推理未来的实例，观察证据并运行标准推理算法。

```
val result = VariableElimination.probability(futureModel.isSpam, true)
println("Probability new email is spam = " + result)
```

期望最大化算法

期望最大化（EM）算法的目标是从数据中学习到一组参数值。该算法依赖于**充分统计量**的概念。为了解释充分统计量，我将回到第 4 章和第 9 章中的β-二项式模型。图 13-9 重现了第 4 章中的模型。

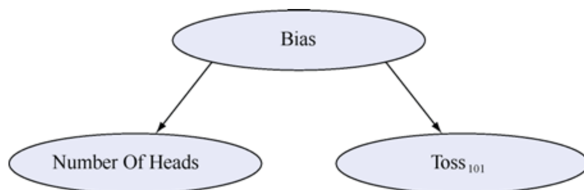


图 13-9 β-二项式模型

我们来复习一下这个模型。

- 模型中有一个表示连续参数的变量，用β分布建模。例如，这个变量可能表示硬币的偏差。β分布有两个参数：α和β。因为它们是参数的参数，有时候被称为**超参数**。
- 另一个变量表示使用上述参数进行给定数量尝试获得成功的次数，用二项分布建模。例如，这个变量表示硬币的偏差由β参数表示时，投掷硬币结果为正面的次数。

您应该还记得，在β参数的先验分布中，α表示观察到任何数据之前想象的成功次数加 1。而β表示想象的失败次数加 1。如果观察到 N_S 次成功，N_F 次失败，参数的后验分布为 Beta(α+ N_S, β+ N_F)。表示参数 MAP 值的分布模型为：

$$\frac{\alpha + N_S - 1}{\alpha + N_S + \beta + N_F}$$

精确记忆上述公式并不重要，重要的是发现在知道成功次数 N_S 和失败次数 N_F 之后，您就有了计算后验参数分布和 MAP 参数值的所有信息。成功和失败发生的顺序无关紧要。因此，N_S 和 N_F 被称为这个β分布的**充分统计量**，因为它们是足以计算后验概率的数据摘要统计量。因为二项分布通常观察 N_S 和 N_F，观察二项分布的结果，可以提供β参数的充分统计量。

参数分布及其充分统计量的例子很多。Figaro 仅包含少数的例子，但是很容易扩展。最常见的例子如下。

- 您已经了解的 Beta 和 Binomial。
- Beta 和 Flip；每个 Flip 为β参数提供一次成功或者失败，如果有许多依赖于某个β参数的 Flip，可以像二项分布中那样，将所有成功和失败加起来。
- Flip 让您在两种结果（真和假）中选择，而 Select 在多个结果中选择，例如 Select(0.2 -> 1, 0.3 -> 2, 0.5 -> 3)。正如 Flip 可以有一个β参数，Select 也可以使用**狄利克雷**参数。除了允许多种结果，每个可能结果有一个超参数之外，狄利克雷参数与β参数类似。和β分布一样，狄利克雷分布的一个参数表示看到某种选择的次数加 1。例如，Dirichlet(2, 4, 3)可能是在 3 个选项中选择一个的先验分布。与之对应的是：想象看到第 1 种结果 1 次，第 2 种结果 3 次，第 3 种结果 2 次。给定狄利克雷参数 d，可以使用 Select(d, List(1, 2, 3))创建由 d 参数化的 Select，其中的可能结果为 1、2 或者 3。

- 表示其他正态分布均值的正态分布。例如，假定您有一个定义为 Normal(2, 1)的变量 m，表示其他正态分布的均值。其他分布均由 Normal(m, 0.5)定义。假设您有其他各个分布的观测值。这些观测值的平均值足以确定 m 的 MAP 值——就等于该平均值。因此，在这种情况下，观测值的平均值是 m 的充分统计量。

基本原理很简单，如果您知道充分统计量，就可以计算 MAP 参数值。但是很遗憾，在大部分情况下，您无法直接观察数据实例，所以不知道充分统计量。

另一方面，如果有定义良好的概率模型，可以使用概率推理计算通常必须观察的变量的期望值。在β-二项分布中，您可以计算预期的成功和失败次数。在正态-正态分布中，您可以计算法线平均值的期望值。这些期望值称为**预期充足统计量**。而且，给定预期充足统计量，您就可以计算 MAP 参数值。遗憾的是，这仍然不够好，因为您没有参数值，所以难以计算预期充足统计量。

总结上述情况：

- 如果只知道预期充足统计量，您可以用公式计算 MAP 参数值。
- 如果只知道 MAP 参数值，可以通过推理计算预期充足统计量。

看起来您似乎陷入了困境，但是这两个因素暗示了一种解决方案，该方案在 EM 算法中得到了体现。该算法的流程图如图 13-10 所示。首先，该过程以对参数值的随机猜测开始。之后循环进行的两个步骤是该算法名称的由来：在 E（期望）步骤中，您使用当前参数值通过概率推理计算预期充足统计量；在 M（最大化）步骤中，您使用计算得到的预期充足统计量计算 MAP 参数值。重复这两个步骤直到预先指定的最大迭代数或者参数收敛。收敛意味着后续的两次迭代得到的参数值几乎完全相同；如果发生这种情况，您就知道算法不会有太大的进展，可以停止。



图 13-10　期望最大化算法流程图

使用 EM 的考虑因素

要说明运行 EM 直到收敛就能得到参数值空间中的最大值并不困难。但是这可能是个**局部最大值**；返回的参数值比附近任何参数的后验概率更高，空间中其他位置的参数值可能有更高的概率。因为不同的初始猜测，EM 可能在不同迭代中返回不同的结果。在实践中，这意味着您可能应该用不同的随机初始猜测多次运行 EM，观察哪一种猜测得到最佳的结果。这种技术称作**随机重启**。

我将 EM 称作**元算法**，是因为它依赖另一种算法进行 E 步骤中的推理。这就是 EM

可以包含任何推理算法、Figaro 提供多种选项的原因。EM 几乎将所有时间花费在 E 步骤中，所以选择合适的推理算法很重要。选择推理算法的考虑因素和单独数据实例上的常规推理相同。适合于在特定模型上进行常规推理的算法也适合于使用 EM 的学习。在电子邮件模型中，您可以使用 VE 进行每个单独数据实例的精确推理，所以可以使用 EM 的 VE 变种。

但是，EM 通常比单一数据实例上的推理慢很多。首先，在每次 EM 迭代中，您必须在所有数据实例上运行推理，这些实例的数量可能很大。其次，您必须运行多次迭代。第三，您可能需要使用随机重启多次运行 EM。

应该运行多少次迭代？这很大程度上取决于所解决的问题，无法给出通用的答案。但是作为经验法则，我有时候发现在 10 次迭代之后回报开始递减，这时最好是以不同的初始猜测重启算法。

EM 在大数据集上可能产生的问题是内存需求，同时在所有数据实例上运行 EM 要求在内存中保存所有实例，包括用于推理的数据结构。这在因子分解算法中尤其成问题，因为因子可能是很大的数据结构，但是，对于抽样算法这也是个问题。因此，对于大数据集，可能会出现内存不足的现象。

在线 EM

为了避免内存问题，Figaro 提供了一种替代方案：在线期望最大化算法。这种算法不同时在所有数据实例上运算，在每次迭代中对所有实例执行 E 步骤，而是逐个计算数据实例。它在单个数据实例上执行 E 步骤，然后将该实例得出的预期充足统计量累加到之前的实例中得到的累计统计量，执行 M 步骤。

一般来说，您将遍历所有数据实例，逐个对这些实例应用 EM 算法。但是也可能碰到这样的情况：您拥有持续的数据流，希望不断从输入数据中学习。在任何时点，都可以使用目前学习到的参数进行有关新实例的推理，继续之后的学习。下面是说明如何在 Figaro 中使用在线 EM 的代码。

程序清单 13-1 在线 EM

```
val parameters = ModelParameters()
val d = Dirichlet(2.0,2.0,2.0)("d",parameters)

class Model(parameters: ParameterCollection, modelUniverse: Universe) { //
  val s = Select(parameters.get("d"), 1, 2, 3)("s", modelUniverse) //
}

def f = () => {
  val modelUniverse = new Universe
  new Model(parameters.priorParameters, modelUniverse)
  modelUniverse
}
```

使用ModelParameters模式，以便在相同模型中使用先验和后验参数

Model 类以参数集和宇宙作为参数，实现最大的灵活性

每个数据实例有自己的宇宙。函数 f 没有参数，负责生成这一宇宙

```
val em = EMWithVE.online(f, parameters)
em.start()

for (i <- 1 to 100) {
  val evidence = List(NamedEvidence("s", Observation(1))) //
  em.update(evidence)
}

val futureUniverse1 = Universe.createNew() //
val futureModel1 = new Model(parameters.posteriorParameters, futureUniverse1) //
println(VariableElimination.probability(futureModel1.s, 1))

for (i <- 101 to 200) {
  val evidence: List(NamedEvidence("s", Observation(2))) //
  em.update(evidence)
}

val futureUniverse2 = Universe.createNew() //
val futureModel2 = new Model(parameters.posteriorParameters, futureUniverse2) //
println(VariableElimination.probability(futureModel2.s, 1))
```

在线EM的第一个参数是生成该宇宙的函数，第二个参数是ModelParameters

对每个数据实例，创建一个命名证据列表，包含该实例的证据（可能通过读取某个文件建立）

用该证据调用在线EM的更新方法

用该证据调用在线 EM 的更新方法

推理之后，您可能遇到更多训练数据，可以交替进行任意频度的训练和推理

完成学习时，可以用后验参数创建一个模型，以通常的方式进行有关该模型的推理

运行上述代码得到如下结果：

```
0.9805825242718447
0.4975369458128079
```

在第一次推理时，训练数据可能是 100 个 1，所以学习到的生成 1 的概率接近于 1。然后在训练数据中增加了 100 个 2，因此现在学到的生成 1 概率接近于 1/2。

学习模型结构

本节的重点是从数据中学习模型参数。我假定您知道模型的结构（变量、函数形式和依赖性），但是不确定数值参数。但是，如果不知道模型的结构该怎么办?

在理想状况下，您已经从数据中学到了模型的结构和参数。遗憾的是，结构学习很难，因为可能程序结构的空间很广阔。想象一下这种情况：在已知某个程序输入输出范例的情况下想要知道程序是如何将输入转换为输出的。这种问题称作**程序归纳**，尽管经过了多年的研究，在这方面仍然没有可靠的通用方法。

那么，如果您不知道模型的结构，可以做什么呢？最佳的方法是在模型中明确地编码有关结构的不确定性。例如，如果不知道模型中是否有某个变量，创建包含和不包含该变量的两种模型版本。如果不知道某个依赖性的方向，同样可以在模型中编码这两种可能性。如果不知道变量是正态分布还是均匀分布，也可以编码这两种依赖性。在所有情况下，都要增加一个变量，表示哪一个模型是正确的。这个变量称作结构变量。

如果试图在单独模型中编码各种可能性的组合，最终会得到大量的模型。幸运的是，许多结构决策是独立的。例如，模型是否包含某个变量和另一个变量是正态分布还是均匀分布可能是相互独立的。其中的诀窍就是在尽可能紧凑的模型中编码尽可能多的结构灵活性。

在模型中表现了多种备选结构之后，通过概率推理进行决策。运用推理，可以确定结构变量的后验概率。您也可以使用 MPE 推理确定所有结构变量的最可能值。然后，使用最可能值将模型归纳为用于未来数据实例的单一模型。

13.3 进一步应用 Figaro

祝贺您！如果您通读了本书，就已经学到了许多关于概率建模和概率编程的知识，包括建模和推理的基础知识、各种建模范式和多种推理算法及其用法。本书的介绍行将结束，但是如果您想要深化对概率编程和 Figaro 使用的认识，可以尝试如下的任务。

- 创建自己的原子元素类库。您可能想要使用当前 Figaro 库没有提供的一些概率分布。添加自己的概率分布并不困难。您可以研究 com.cra.figaro.library.atomic 中的元素类定义（不管是连续元素还是离散元素），以得到相关的概念和实现方法。
- 创建自定义复合元素。这比创建原子元素类更容易。如果发现自己编写和重用以一些元素为参数、返回其他参数的函数，可以考虑将其转换为复合元素类。这将使它们更容易在任何模型中使用。同样，您可以从 com.cra.figaro.library.compound 中的定义找到思路。
- 研究算法的调试版本。大部分 Figaro 算法提供名为 debug 的标志，该标志通常设置为 false。如果将其设置为 true，它通常会提供详尽的输出，这些输出可以帮助您找出算法没有按照预期运行的原因。当然，理解调试信息需要对算法所做的运算有一定的了解，但是本书提供的基本信息应该已经足够。例如，变量消除法的调试输出显示在解题过程中任何时点创建的所有因子。当您调试自己的模型时，将发现为所有元素命名是很有益的。
- 如果您使用 Metropolis-Hastings 算法，可能发现 Metropolis-Hastings.test 方法很有用。在这个方法中，您用 Element.set 方法设置任何需要的变量值，将系统设置为特定状态。然后，可以测试以观察从该状态运行 MH 产生结果的可能性。您可以指定结果状态所能满足的预测数量，并找出满足各种预测的次数占比。

- 使用宇宙的概念帮助解决困难的推理工作。例如，您可以将模型分解为两个宇宙，在其中一个宇宙上执行模拟退火算法，搜索一组变量的最可能值，在另一个宇宙上计算从这些变量值生成的证据概率。实际上，变量消除法已经采用了这种概念，在相关的宇宙中使用不同算法计算证据概率，但是使用多个宇宙的思路是通用的思路。Figaro 团队正在升级宇宙框架，提供统一的算法结构，使同一个算法能够在多个宇宙上运行，无缝地组合不同宇宙中的算法，所以随着后续发行版本的推出，使用这种技术的潜力将有增无减。

如果您对提升 Figaro 的实用性有想法，请您考虑在我们的 GitHub 网站（https://github.com/p2t2/figaro）上分享。我们欢迎您做出贡献，也始终乐意回答任何问题。感谢阅读本书，希望您能好好地利用概率编程。

13.4 小结

- 您可以如下方式进行动态概率模型的推理：将其展开到固定数量的时间步，或者使用过滤算法进行随时监控。展开的方法使您可以向前和向后推理，而过滤使您可以自己决定推理的时长、节约内存。
- 您可以使用任何标准推理算法进行贝叶斯学习，学习参数值上的后验概率分布，并用这一后验分布预测新情况。
- Figaro 的 ModelParameters 结构有助于 MAP 学习，这种学习通过期望最大化（EM）算法执行，该算法包装标准的推理算法。
- 在线 EM 在数据实例很多，内存可能成为问题或者希望从持续的数据流中学习时很有用。

13.5 练习

在 www.manning.com/books/practical-probabilistic-programming 可以找到部分练习的答案。

1．工厂制造的 1000 辆汽车中有一辆出现故障。在任何给定的日子里，汽车的燃油效率或高或低，遵循如下分布：如果前一天的燃油效率很低，则当天燃油效率低的概率为 90%。如果前一天燃油效率很高，则对于故障车辆来说当天燃油效率低的概率为 90%，而对于正常车辆来说为 5%。您的车辆有一个燃油经济性仪表，可以显示车辆当天燃油效率的高低。这个仪表显示正确值的概率为 0.95，显示不正确值的概率为 0.05。

a）画出表示该系统的动态贝叶斯网络。

b）编写一个 Figaro 程序表示这个系统。

c）从这个程序创建两个长度为 100 的观察序列，一个用于正常车辆，另一个用于故障车辆。

d）使用粒子过滤监控该系统状态。在两个观察序列上运行粒子过滤，使用 100 个样本。您可能发现在大部分时候无法检测出故障车辆。您认为这是什么原因？

e）现在用 10000 个样本重复（d）。通常您将得到不同的结果，您能解释原因吗？（一定要多次运行试验——粒子过滤不总能产生相同的答案）

2．现在，对前一个练习的模型稍加更改，使粒子过滤运行得更顺畅。故障车辆不总是保持故障状态；每个时间步在一定概率下，它能够自行修正。同样，正常车辆可能发生故障。具体地说，从一个时间步到下一个时间步车辆保持相同故障状态的概率为 0.99，状态出现变化的概率为 0.01。

用这个模型在 100 个样本上运行粒子过滤。新结果和原来的结果有何不同？用 10000 个样本重复试验。评估这一平衡措施。

3．这个练习指出 EM 算法的一个陷阱。考虑图 13-11 所示的简单贝叶斯网络，您将学习该网络的参数。

a）创建 Figaro 程序，用模型参数学习模式表示该网络。用 Beta(1,1)作为所有参数的先验分布。

b）创建有 10 个实例左右的训练集，仅观察变量 X 和 Z。确保在每个训练实例中，X 和 Z 取值相同。有时候它们都为真，有时都为假。

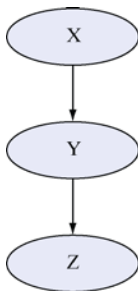


图 13-11　练习 3 的贝叶斯网络

c）使用 EM 算法学习模型参数。试验 EM 的不同变种。

d）现在用后验参数创建一个测试案例。观察 X 为真，并计算 Z 为真的概率。您可能发现，尽管 X 为真，训练集中的 X 和 Z 总是相等，但是这个概率并不接近 1。试解释其中的原因。

e）下面我们来解决这个问题。将已知 X 为真时 Y 的先验分布改为 Beta(2,1)，Y 为真时 X 的先验分布为 Beta(1,2)。同样，使用 EM 学习参数并运行测试。这一次，Z 为真的概率应该接近于 1。您认为为什么会出现这样的结果？当您自己使用 EM 时，可以从

中得到什么启示？

4．现在重复问题 3，用贝叶斯学习代替 EM。

a）在这个问题上尝试不同的推理算法。哪种算法效果最好？您认为原因是什么？

b）对比贝叶斯学习和 EM 的结果。对所有变量使用 Beta(1,1)先验分布或者对 Y 使用 Beta(2,1) 和 Beta(1,2)，贝叶斯学习有没有显著的差异？使用 Beta(2,1)和 Beta(1,2)先验分布是否在 EM 中造成显著的不同？如果有，您认为是什么原因？

5．您已经看到参数学习的工作原理，现在可以回顾第 3 章中的垃圾邮件过滤示例了。

a）重现该示例，使用模型参数学习模式。

b）将学习方法从第 3 章中的基本 EM 方法（同时在所有电子邮件上训练）改为离线学习方法。

附录 A　获取和安装 Scala 和 Figaro

本附录包含安装 Figaro 并在您的计算机上运行的指南。目前为止最简单的方法是使用 Scala 构建工具（sbt）。用 sbt 运行 Figaro 的说明在 A.1 小节中描述。如果因为某种原因无法或者不想使用 sbt，可以参考 A.2 小节中介绍的人工安装方法。

本书代码使用的 Figaro 官方版本是 3.3。早期版本的 Figaro 不支持本书描述的所有功能。我们尽可能使未来的版本向后兼容，所以您应该也可以使用更新版本的 Figaro。

Figaro 3.3 使用 Scala 2.11.x。遗憾的是，Scala 开发人员不保持版本间的向后兼容性，将 Figaro 移植到新版本 Scala 需要花费时间，所以 Figaro 不总能在最新的 Scala 版本上工作。sbt 负责这些依赖性，确保使用正确的 Scala 版本。

A.1　使用 sbt

运行本书中代码的最简单方法是使用 sbt。如果下载本书的代码，它们是预先打包为一个 sbt 项目的。您所需要做的就是从 www.scala-sbt.org 下载和安装 sbt。然后，在本项目中运行 sbt 时，它将自动拉进相关的 Scala 和 Figaro 版本，所以您不需要安装任何其他软件。sbt 还确保项目配置正确的类路径（classpath），避免一些麻烦。

使用本书所需要知道的 sbt 命令如下：

```
sbt console
sbt "runMain object arguments"
```

运行交互式 Scala 命令解释程序

以给定的命令行参数运行指定对象中的主方法。对象必须加上包名，例如 chap01.HelloWorldFigaro

要运行这些命令，需要导航到 PracticalProbProg/examples 目录，这是项目的顶级目录。

注意：如果已经在 sbt 面板中，就不需要再次输入 sbt，可以输入 runMain object arguments。

sbt 有许多功能，也是构建和运行代码的好方法。您可以在我们的项目中使用 Build.scala 作为起点。sbt 也可以和 Eclipse 一起使用。

A.2　在没有 sbt 的情况下安装和运行 Figaro

虽然 sbt 是一种实用的工具，但您可能想要以不同的方式管理自己的工作空间。本安装指南的余下部分描述在没有 sbt 的情况下安装和运行 Figaro 的方法。

运行 Figaro 首先需要 Scala。Scala 编译程序可以从命令行或者集成开发环境（IDE）运行。支持 Scala 开发的 IDE 包括 Eclipse 和 IntelliJ IDEA。NetBeans 也有一个 Scala 插件，但是似乎不能支持 Scala 的最新版本。在下面的指南中，我将说明如何获取 Scala 和 Figaro，以及从命令行运行使用 Figaro 的 Scala 程序。我不提供特定 IDE 的使用说明。如何包含 Figaro 库的细节请参见 IDE 和 Scala 插件文档。

1．由 Scala 入手，从 http://scala-lang.org/download/下载合适于 Figaro 版本的 Scala 版本。根据 http://scala-lang.org/download/install.html 的 Scala 安装指南，确保您可以运行、编译和执行文档中提供的 Hello World 程序。

2．下一步是获取 Figaro。Figaro 二进制分发程序托管在 Charles River Analytics 公司的网站。访问 www.cra.com/figaro，确保使用与 Scala 版本相匹配的 Figaro 版本。每个可用的下载链接都是一个包含 Figaro JAR（JAR 是编译字节代码的 Java/Scala 格式）、示例、文档、Scaladoc 和源代码文件的压缩包。在分发版本中，Figaro JAR 文件名以“fat”结束，表示这是包含所有运行 Figaro 必要文件的 JAR。单击合适的链接，然后解压下载的压缩文件。

3．[可选]将 Figaro JAR 的完整限定路径名添加到 classpath。这可以通过将 Figaro JAR 添加到操作系统中的 CLASSPATH 环境变量来完成。编辑 CLASSPATH 的过程在不同操作系统中各不相同。您可以在 http://docs.oracle.com/javase/tutorial/essential/environment/paths.html 查看关于 PATH 和 CLASSPATH 环境变量的细节。

a）如果 CLASSPATH 不存在，创建一个。我总是喜欢在 CLASSPATH 中包含当前工作目录，所以将 CLASSPATH 设置为“.”。

b）现在，在 CLASSPATH 中添加 Figaro JAR。例如，在 Windows 7 上，如果 figaro_2.11-3.3.0.0-fat.jar 在 C:\Users\apfeffer 文件夹中且 CLASSPATH 目前等于“.”，将 CLASSPATH 更改为 C:\Users\apfeffer\figaro_2.11-3.3.0.0-fat，用对应的 Scala 版本号代替 2.11，用对应的 Figaro 版本代替 3.3.0.0。

4．现在，您可以像任何 Scala 程序一样编译和运行 Figaro 程序。将如下的测试程序放在名为 Test.scala 的文件中。首先，假定您按照第 3 步并更新 CLASSPATH。

a）如果从包含 Test.scala 的目录中运行 scala Test.scala，Scala 编译器将首先编译该

程序，然后执行。该程序应该输出 1.0。

b）如果运行 scalac Test.scala（注意 scalac 最后的 c），Scala 编译器运行并生成.class 文件。然后，在同一个目录运行 scala Test 执行程序。

c）如果没有按照第 3 步执行，可以使用-cp 选项从命令行设置 CLASSPATH。例如，要编译和执行 Test.scala，假定 figaro_2.11-3.3.0.0-fat.jar 在 C:\Users\apfeffer 文件夹中，可以运行 scala –cp C:\Users\apfeffer\ figaro_2.11-3.3.0.0-fat Test.scala。

下面是测试程序：

```
import com.cra.figaro.language._
import com.cra.figaro.algorithm.sampling._

object Test {
  def main(args: Array[String]) {
    val test = Constant("Test")
    val algorithm = Importance(1000, test)
    algorithm.start()
    println(algorithm.probability(test, "Test"))
  }
}
```

上述程序应该输出 1.0。

A.3 从源代码编译

Figaro 在 GitHub 上作为开源项目维护。这个 GitHub 项目是 Probabilistic Programming Tools and Techniques（概率编程工具与技术，P2T2），位于 https://github.com/p2t2。P2T2 目前包含 Figaro 源代码，但是我们计划更新它，包含更多的工具。如果您想要自己查看源代码和构建 Figaro，请访问我们的 GitHub 网站。

Figaro 使用 sbt 管理构建。要从 GitHub 源代码构建 Figaro，建立您的 GitHub 账户存储库的一个分支，然后使用 Git 的克隆功能，从 GitHub 账户将源代码下载到您的机器：

```
git clone https://github.com/[your-github-username]/figaro.git
```

该项目有多个分支；访问“master”分支获得最新的稳定版本，也可以访问最新的“DEV”分支获得更尖端的功能（这是正在推进中的工作，因此较不稳定）。下载和安装 sbt，启动程序进入命令提示行，按照顺序输入如下命令：

```
> clean
> compile
> package
> assembly
> exit
```

这就为相关的 Scala 发行版本创建了一个 Figaro 版本，您可以在“target”（目标）目录中找到这些工件。

附录 B　概率编程系统简况

正在开发中的概率编程系统（PPS）越来越多。在本附录中，我将简短地介绍广泛使用的系统及其关键功能。在可能的情况下，我还提供下载这些系统的 URL。在此，我不想涵盖所有的系统；为此我要向未被提及的系统的开发者们道歉。我还要预先为系统说明中的任何错误或者重大遗漏道歉。

PPS 的特性可以从几个维度描述：

- 语言的表达能力如何？例如，它是否支持用户定义函数、无向模型、离散和连续变量、开放宇宙模型和任意数据类型的变量？
- 系统采用何种部署策略？它是否提供独立语言、现有语言中的库或者包含概率扩展的现有语言新实现？如果是独立语言，是否提供来自现有语言的接口？
- 它使用哪一类编程风格，如函数式、逻辑式、命令式或者面向对象？
- 系统提供哪种类型的推理算法，如分解和抽样算法？是否支持动态推理？
- 支持哪些类型的查询？

下面是 PPS 的一些例子。

BUGS (WWW.MRC-BSU.CAM.AC.UK/SOFTWARE/BUGS/)

BUGS 是 Bayesian Inference Using Gibbs Sampling（使用 Gibbs 抽样的贝叶斯推理）的缩写，顾名思义，它是围绕称作 Gibbs 抽样的马尔科夫链蒙特卡洛（MCMC）算法构建的。BUGS 是最早的 PPS 之一，已经在社会科学等学科中流行起来。在表现能力方面，BUGS 不允许用户定义函数，主要关注连续变量，但是为变量提供了广泛的分布。BUGS 是以独立语言的形式实现的。

STAN (HTTP://MC-STAN.ORG/)

Stan 是一种流行的概率编程系统，尤其是在统计学的圈子内，它能够执行许多类统计推理。Stan 的主要推理算法是 MCMC 的高效形式。和 BUGS 一样，Stan 专注于连续变量，提供广泛的分布类型。Stan 是一种独立语言，但是提供了流行语言（如 R、Python、

MATLAB）的接口。

FACTORIE (HTTP://FACTORIE.CS.UMASS.EDU/)

FACTORIE 是一种在自然语言处理应用上取得许多成功的概率编程系统。和大部分其他 PPS 不同，FACTORIE 使用命令式风格，明确地创建因子图，在这些图上可以执行 MCMC 等算法。和 Figaro 类似，FACTORIE 是一个 Scala 库。

PROBLOG (HTTPS://DTAI.CS.KULEUVEN.BE/PROBLOG/)

和本附录中介绍的其他 PPS 不同，ProbLog 基于逻辑编程。如果您喜欢使用 Prolog 等逻辑编程语言，ProbLog 可能是个很好的选择。ProbLog 可以视为 Prolog 的概率扩展。概率逻辑编程的基本思路是有一组基本事实和一组逻辑规则，其他派生的事实均遵循这一组规则。任何派生事实的概率是一组基本事实的概率，因此派生事实由这些基本事实产生。ProbLog 仅限于离散变量。对于推理，ProbLog 使用证明推导等逻辑编程技术。

BLOG (HTTPS://SITES.GOOGLE.COM/SITE/BLOGINFERENCE/)

BLOG（Bayesian Logic，贝叶斯逻辑）是逻辑和函数式 PPS 的一种混合物。BLOG 中的陈述类似于逻辑陈述，但是 BLOG 模型中有一个生成流程，表示可能世界的生成方式，这与 Figaro 等 PPS 类似。BLOG 采用开放宇宙建模，在这种方法中，您不知道对象的数量和身份，如果您有这类模型，BLOG 就是好的选择。BLOG 使用 MCMC 进行推理，是一种独立的语言，但是您可以用 Java 编写自定义提议方案。

CHURCH (HTTPS://PROBMODS.ORG/PLAY-SPACE.HTML)

Church 是基于类 Lisp 语言 Scheme 的函数式 PPS。在表现能力上和 Figaro 类似，它也支持复杂控制流和递归，以及丰富数据结构，但是它不是面向对象的。Church 有多种实现。上述的 URL 指向 WebChurch，这个 Web app 包含一个很好的交互式概率编程教程。

ANGLICAN (WWW.ROBOTS.OX.AC.UK/～FWOOD/ANGLICAN/)

Anglican 是相对新颖的语言，在表现能力上类似于 Church，其主要特征是高效和准确的抽样算法。

VENTURE (HTTP://PROBCOMP.CSAIL.MIT.EDU/VENTURE/)

Venture 是来自一些 Church 开发者的新语言。Venture 的主要创新是推理编程，为用户提供了表现力强、细粒度的交互推理控制，主要使用抽样算法。

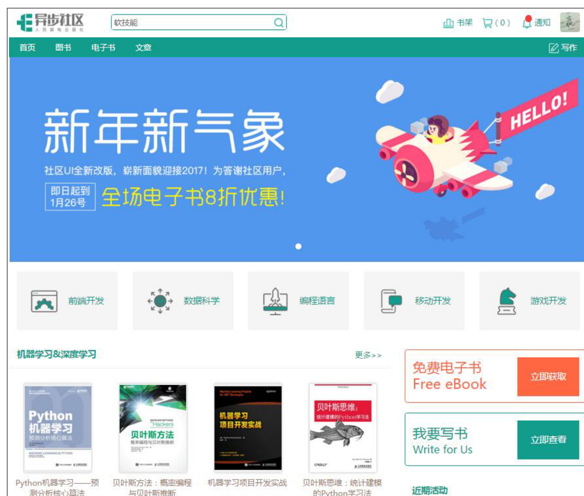
DIMPLE (HTTP://DIMPLE.PROBPROG.ORG/)

Dimple 是 Gamalon 生成的 PPS。尽管它能够表现离散和连续变量以及有向和无向模型，但是仅限于有限的固定结构模型。对于这些模型，Dimple 提供一些高效的分解推理算法。

异步社区
人 民 邮 电 出 版 社
www.epubit.com.cn

欢迎来到异步社区！

异步社区的来历



异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 0 使用积分 里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信服务号



微信订阅号



官方微博



QQ 群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿 & 咨询：contact@epubit.com.cn

概率编程实战

你所积累的有关客户、产品和网站用户的数据不仅能帮助你解读过去的情况，还可以帮助你预测未来！概率编程使用代码从数据中进行概率推理。通过应用特定的算法，你的程序可以确定不同结论的概率。这意味着你可以预测未来事件，如销售趋势、计算机系统故障、试验结果和其他许多重要的关注点。

本书向程序员们介绍概率编程。在书中你能立刻接触到实际的示例，如构建垃圾邮件过滤器、诊断计算机系统数据问题、恢复数码图像。你会发现概率推理中的算法能够帮助你做出有关社会化媒体使用情况等问题的长期预测。在这本书中，你还将学习用于文本分析的函数式编程，预测微博传播等社会化现象的面向对象模型，以及估算现实生活中社会化媒体使用量的开放宇宙模型。本书还包含了介绍概率模型如何帮助决策和动态系统建模的章节。

本书主要内容

- 概率建模入门
- 用 Figaro 编写概率程序
- 构建贝叶斯网络
- 预测产品生命周期
- 决策算法

本书假定读者之前没有接触过概率编程。懂得 Scala 的知识对阅读本书是有帮助的。

Avi Pfeffer 概率编程的先驱，Figaro 概率编程语言的首席设计者和开发者。他一直致力于 Figaro 在多个问题上的应用，包括恶意软件分析、汽车健康监控、气象模型建立和工程系统评估。

“概率编程从研究实验室走向现实世界的重要一步。”

—— 摘自本书序言，Stuart Russell，加州大学伯克利分校

“清晰的示例和对困难、复杂主题的详尽解释。”

—— Mark Elston，Advantest America

“连贯、实用、易于理解。这是一本不可思议的 Scala 概率编程实战书籍。”

—— Kostas Passadis，IPTO

“概率编程真是复杂！但是 Avi 使这一主题变得简单直观，易于学习。”

—— Earl Bingham，Eyelock







异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn


ISBN 978-7-115-44874-3

美术编辑：董志桢

分类建议：计算机 / 程序设计
人民邮电出版社网址：www.ptpress.com.cn